UNIVERSIDAD CARLOS III DE MADRID

TESIS DOCTORAL

New Contributions for Modeling and Simulating High Performance Computing Applications on Parallel and Distributed Architectures

Autor:
Alberto Núñez Covarrubias

Directores:
Jesús Carretero Pérez
Javier Fernández Muñoz

DEPARTAMENTO DE INFORMÁTICA

Leganés, Enero de 2011

TESIS DOCTORAL

New Contributions for Modeling and Simulating High Performance Computing Applications on Parallel and Distributed Architectures

Autor: Alberto Núñez Covarrubias
Director: Jesús Carretero Pérez
Co-Director: Javier Fernández Muñoz

Firma del tribunal calificador:

Presidente:	Firma:
Vocal:	Firma:
Vocal:	Firma:
Vocal:	Firma:
Secretario:	Firma:

Calificación:

Leganés, a __ de _ _ _ _ _ _ _ _ _ _ de _ _ _ _.

A journey of a thousand miles
must begin with a single step.

– Lao Tzu

In the middle of difficulty lies opportunity.

– Albert Einstein

Agradecimientos

Bueno, una vez finalizada esta tesis, me gustaría dedicar un pequeño espacio para agradecer a la gente que ha estado conmigo, que me ha ayudado, que me ha apoyado (lo que más), que me ha soportado/aguantado (lo que menos), y que en algunos casos ha sufrido conmigo mientras la escribía (lo que menos aún).

Principalmente esta tesis está dedicada a mis padres Pilar y Manuel, los cuales han demostrado siempre un apoyo incondicional hacia mí. Gracias a ellos y a los sacrificios que llevan haciendo desde que tengo uso de razón (que son unos pocos), estoy ahora donde estoy. Escribir esta tesis ha sido un paseo comparado con lo que habéis hecho durante estos 30 años. Solo espero que cuando yo sea padre (si algún día lo soy), sea la mitad de bueno de lo que lo habéis sido vosotros conmigo. Sin duda para poder expresar mi gratitud necesitaría el espacio equivalente a varias tesis como esta, pero os tendréis que conformar con este pequeño párrafo. ¡Gracias!

A mis directores de tesis, Jesús Carretero y Javier Fernández.

A Lorena, por tu apoyo en la recta final y en la parte más difícil, por hacer que sea un poco más fácil, por animarme y por estar ahí siempre con una sonrisa. No sé si hubiera terminado ahora de no ser por ti. ¡Gracias!

A mi hermano Sergio (alias enano) y a Rosy. Enano, hace 5 años te lo dije, y te lo vuelvo a repetir ahora (cansino). Espero que dentro de unos años estés escribiendo algo parecido a lo que estoy escribiendo yo ahora, ¡tolai! PD: ¡Despierta!

No puede faltar el tolai de mi primo Manuel (alias Atete). Aunque rabias más que el tío Gilito, y me has dado más hostias que a un saco de boxeo, te voy a poner en los agradecimientos. Es muy posible que no hubiera escrito esta tesis de no ser por ti. Y como aún te debo unas pocas de copas, así me quitas alguna. También mis tíos Antonio y Sagrario, y mi prima Juana se merecen su hueco aquí.

También quiero dedicar una parte de esta tesis a las personas que desgraciadamente y por ley de vida, hoy no pueden ver cómo termino esta etapa. En esta vida a cada uno nos tocan 4 abuelos por naturaleza. Yo puedo considerarme muy afortunado por haber tenido hasta 8: 4 abuelos naturales y 4 abuelos postizos. Para aquéllos que hoy no estáis hoy aquí, mis abuelos paternos Manuel e Isabel, mi abuelo materno Primitivo, mis abuelos canarios, Pepe y Lola, y mi abuelo de Castellón, Francisco.

Por supuesto al resto de mis abuelos, mi abuela materna Pilar y mi abuela de Castellón Luisa.

Miguel (alias Mickey), amigo de toda la vida, de los que siempre están ahí, y que ayudó a la traducción de estos mismos agradecimientos. ¡Eres un grande! Por supuesto también agradecer a tus padres Juan Miguel y Felisa, y tu hermana Maria Elena su apoyo

incondicional, tanto conmigo como con mi familia.

A David, otro amigo de toda la vida con el que pasé los primeros años de mi vida, de los que guardo muchísimos buenos recuerdos, y que todavía sigue ahí. Bueno crack, aunque me llames *pesao*, ¡acaba la carrera tío! ;) Por supuesto tus padres Ramón y Julia, y tu hermano Pablo también os merecéis vuestro cachito de agradecimiento.

A Mark, amigo y compañero de incontables heroicas. Hemos estado cogiendo olas un Diciembre en Santander a 4 grados, hemos hecho snowboard con vientos de más de 100 Km/h que nos subían montaña arriba, nos metimos en Portugal en el agua con menos de 2 palmos de profundidad con el fondo de roca lleno de erizos; Aún recuerdo cuando me quitaste las púas de la rodilla en el agua, o cuando caminabas como un pingüino por llevar la planta del pie llena de púas. Y así podría seguir hasta llenar varias páginas. Mark, ¡boarding till the edge of times!

No me olvidaría nunca de mis compañeros de la Escuela Politécnica. Hace 12 años que entramos en la escuela, y a día de hoy seguimos quedando todos los años para reunirnos, reírnos y acordarnos de los tiempos de estudiantes. Algunos ya casados, otros con hijos ¡No corráis tanto que me hacéis mayor! Y eso que soy de los más jóvenes del grupo. Para todos vosotros también hay un pequeño espacio aquí: Andrés, Roberto, Jose, Víctor, Victoria, Darta, Ray, Alba, Juan, Eva, Raúl, Fernando, Guillermo, Hugo (de infiltrado) y Cía.

A mi amigo Antonio Cuevas. ¡Qué te voy a decir a ti! Apoyo infinito, *and beyond*. Salimos en el Oktober Fest, y en el otro de Stuttgart. Fuimos a esquiar a Austria el día antes de un charlote (*unbelievable*). Hemos compartido momentos muy buenos y de los malos también, como tiene que ser. En fin Antonio, no tengo palabras. Un millón de gracias por todo.

A Patrick y Julie. Vosotros también tenéis el hueco aquí. Patrick, ¡tienes que mejorar tu snowboard! La próxima en Suiza.

Oscar García, no puedes faltar en los *greetings*, aunque ahora andes por tierras canadienses. Compañero de fatigas en Silicon Valley. Descubrimos juntos Santana Row. Recorrimos California en coche. Pues eso, un sinfín de experiencias compartidas en California. El verano de 2009 no hubiera sido lo mismo sin ti. ¡No saben en Silicon Valley lo que tuvieron ahí!

A Félix García, que fue mi tutor del Proyecto Fin de Carrera y en parte gracias a él entré a hacer el doctorado.

A Isabel Muñoz, que fue mi tutora del Proyecto Fin de Carrera en la Escuela Politécnica de Madrid.

A mi tía Eli (mi madrina) a mi tío Jose, a mis primas Sheila y Lorena y al tolai de mi ahijado Carlitos.

A Mario Blaum, por todo tu apoyo durante el verano de 2009. Fue sin duda el mejor verano de mi vida, en Silicon Valley. A veces me cuesta creer que estuviera allí trabajando y cogiendo olas. Siempre me acordaré de las conversaciones y discusiones en el café por la tarde. Muchísimas gracias por todo.

A Steve Heztler y Daniel Smith. Gracias a Steve por darme la oportunidad de hacer un internship en IBM. Fue una experiencia inolvidable.

A Raulas (alias Raul). Como no mi amigo Raúl. Hemos compartido unas pocas de prácticas y muchas horas de ordenador. Y también momentos de los buenos. Y ahora te

metes a hacer un Máster para hacer Terminators. Gracias a ti también por tu apoyo.

A Rauli (alias Raúl Sampedro). Raúl tío, eres un crack, a pesar del juego chungo que hiciste (Star Wars Unleashed II). A ver si nos esforzamos un poquito. ¡Deja el listón alto en el valle!.

A mi prima Isa, por su apoyo constante desde el principio, y antes aún.

A Roberto (alias Rowel). A pesar de que seas un tolai, y estés todo el día gruñendo, también tienes aquí tu hueco. Aunque en cierto modo también has contribuido al dejarnos usar las aulas para validar el simulador. ¡Apunta un café!

A mis compañeros de la universidad Carlos III de Madrid.

A mi amigo *surfero* Pedro (alias Peterpato) y a la cuadrilla palillera P^3: Pablo, Pableras (Recueril) y Paco. También Vela, pero este tiene más estilo, así que le perdonamos. ¡Gracias a todos vosotros también, tolais!

Por supuesto a Efrén (alias frames). Aún me acuerdo cuando estuvimos horas para configurar un módem de 9600 baudios, más grande que la pantalla del ordenador, cuando teníamos 14 años. ¡Y nos conectamos a algo! Este y muchos momentos similares hemos pasado juntos, ¡vaya tiempos!

Yeray (alias bin). Bueno bin, gracias también a ti por muchas cosas. Aunque estamos muy lejos seguimos manteniendo la amistad y el buen humor.

Aaron (alias CoolAce). Bueno Aaron, gracias a ti también canarión de pura cepa, aunque andes ahora por tierras germanas. Espero que te vaya todo bien por ahí, crack.

A Jose y Agustín, juntos revolucionaremos el cloud computing. Gracias por todos esos mensajes de ánimo. Levantarían a cualquiera.

A mi amigos de Getafe, Marcelino, Marta, Javi, María, Carran (alias surfer), Alicia, Javier Calvo, María, David y Cía.

Bueno, una vez leídos los agradecimientos, os dejo con el resto de la tesis. Como dijo una vez Homer J. "Esta máquina de movimiento perpetuo no sirve para nada, se acelera cada vez más". Espero que la disfrutéis.

Acknowledgements

Well, now that I have finished my thesis, I would like to say a few words to thank all of the people who have been with me during this period of time, to all those who have helped and supported me, and especially those who have even shared my suffering whilst I was writing it.

This thesis is mainly dedicated to my parents, Pilar and Manuel, who have always showed me unconditional support. Thanks to them, and all the sacrifices they have made since I was a child, I am now here. Writing this thesis has been a piece of cake compared to all of the things they have done throughout all of these 30. I wish I could be half as good parent as they have been to me. Without a shadow of a doubt I would have to write several theses like this one to express my gratitude to you, but you will have to put up with this small paragraph. Thanks!

Thanks to my thesis advisors Jesús Carretero and Javier Fernández.

Also I must say a big thank you to Lorena for your support in the final and hardest part of this thesis, for making it easier, for encouraging me, and for always being there with a smile on your face. I don't know if I would have been able to finish now without your help. Thanks!

I would like to say thanks to my brother Sergio (a.k.a. enano) and Rosy. Enano, I told you five years ago, and I will tell you again. In a few years, I hope you write something similar to what I am writing right now. Take the hint! PS: So, wake up!)

I can't leave my cousin Manuel (a.k.a Atete) out of this thesis. Although you grumble, for the sake of it, and you cannot never ever stop picking on me, I am going to include you in these greetings too. It's very likely that I would not be able to write this thesis without you. And so, I owe you some drinks, and because I owed you a long list of them already, it will be a good time to make the list shorter. I also want to thank my aunt and uncle Sagrario and Antonio, and my cousin Juana, they deserve a space here.

I am very lucky to have had eight grandparents in my life. I want to dedicate this short but special paragraph to them. Unfortunately, some of them cannot be with me today to share my joy.: my paternal grandparents, Manuel and Isabel, my maternal grandfather, Primitivo, my canary grandparents, Pepe and Lola, and Francisco, my grandfather from Castellón. And of course, I also dedicate it to my two surviving grandparents: my maternal grandmother, Pilar, and my grandmother from Castellón, Luisa.

Miguel (a.k.a. Mickey), lifelong friend who always is there, who also helped me writing these words of thanks in English. Man, you rock!. Of course, I would like to thank your parents Juan Miguel and Felisa, and your sister, Maria Elena. Thanks for the unconditional support you all have shown my family and me.

To David, another lifelong friend. We have grown together since we were babies. I have fond childhood memories of you dude. Anyway, even if you think I am a pain in the neck, finish your degree man! Of course, your parents, Ramón and Julia, and your brother, Pablo, deserve a piece of my gratitude.

To Mark, my friend and surf-mate, who I have lived lots of uncountable epic surfing adventures with. We have practiced body boarding at 4 degrees in Santander in December, we have practiced snowboarding with winds at 62 mph, and also in Portugal, where it took us less than a second to fall into the water which has rock reefs, no less than 30 centimeters deep. Let alone the rock reef was full of urchins. I still remember you removing spines from my knee, and also you walking like a penguin because of the urchin spines stuck in your feet. Mark, boarding till the edge of times!

I will never forget my classmates in the Escuela Politécnica de Madrid. We met twelve years ago for the first time, and we still continue having our Christmas dinner every single year. Some of you are married, some other have kids. Slow down please! You are making me feel old! There are a few words of thanks for you too: Andrés, Roberto, Jose, Víctor, Victoria, Alberto, Darta, Ray, Alba, Juan, Eva, Raúl, Fernando, Guillermo, Hugo & Co.

To my friend Antonio Cuevas. What can I say about you, man? Infinite supporting, and beyond. We were in the 2008 Oktober Fest in Munich, and also in the Sttutgart's one. We practiced snowboarding in Austria (unbelievable). We have shared very good moments, and also bad ones too, such as good friends. In short, I have no words. A million thanks!

To Patrick and Julie. Patrick, you need to improve at snowboarding. Next time, we'll go to Switzerland.

To Oscar García. You cannot be missed out here, even being in the Canadian lands. You were my fellow-sufferer in Silicon Valley. We discovered Santana Row together, and crossed California by car. Anyway, they are an uncountable number of shared experiences in California. The summer of 2009 would not have been the same without you, man.

To Félix García, who was my master thesis advisor and, thanks to him, to a large extent, I started my PhD.

To Isabel Muñoz, who was my master thesis advisor in the Escuela Politécnica de Madrid.

To my aunt (and godmother) Elisa, my uncle Jose, my cousins Sheila, Lorena, and Carlitos.

To Mario Blaum, for all your support during the summer of 2009. It was without a doubt, my best summer ever, in Silicon Valley. I will always remember those talks in the coffee-break at IBM. Thank you very much for everything.

To Steve Hetzler and Daniel Smith. Thanks Steve for giving me the chance to do an internship with you at IBM. It was an unforgettable experience.

To the great Raulas (a.k.a. Raul). My friend, we have shared a lot of hours in front of a computer doing university exercises. We have had good times too. And now, you have joined a Master to create Terminators. Thanks for your support too.

To Rauli (a.k.a. Raul Sampedro). Hey man, you are a one of the best. Despite the mediocre game you made (Starwars Unleashed II). You have to take it seriously! Leave the level so high man!

To my cousin Isa, for her constant support from the beginning, and even before the

beginning.

To Roberto (a.k.a. Rowel). Besides being a green, and being always grunting, you also have a small section here. However, you also contributed to do this thesis by letting us use the empty classrooms to validate our simulator. I owe you one!

To my mates of Universidad Carlos III de Madrid.

To my *snow-surfer* friend Pedro (a.k.a. Peterpato) and the skiing squad P^3: Pablo, Pableras (Recueril) and Paco. Vela too, but he is a stylish man, so, I forgive him. Thanks to all of you, "tolais"!

Of course, my friend Efrén (a.k.a. frames) must be mentioned here too. We were fourteen, but I still remember you and me spending many hours trying to set up a 9600 bps modem. It was bigger than the computer's monitor. Suddenly, we established a connection with something! Lots of good times frames!

To Yeray (a.k.a. Bin). Hey Bin, thanks for so many things. Although we are so far away from each other, we still maintain our friendship and our good sense of humour.

To Aaron (a.k.a. CoolAce). Well Cool, despite of being in German lands now, you are a Canarian through and through. Thank you too! Keep on rollin'.

To Jose and Agustín, we will revolutionize cloud computing. Thanks for all those encouraging messages.

To my friends from Getafe: Marcelino, Marta, Javi, María, Carran (alias surfer), Alicia, Javier Calvo, María, David & Co.

Well, once you have read all of these sincere words of gratitude, I leave you with the rest of this thesis. Just like Homer J. said, "And this perpetual motion machine she made today is a joke! It just keeps going faster and faster". Enjoy!

Resumen

En esta tesis se propone una nueva plataforma de simulación específicamente diseñada para modelar sistemas paralelos y distribuidos, la cual se basa en la integración del modelo de los cuatro sistemas básicos en una única plataforma de simulación. Estos sistemas están formados por el sistema de almacenamiento, el sistema de memoria, el sistema de procesamiento (CPU) y el sistema de red. Las principales características de esta plataforma de simulación son flexibilidad, para abarcar el mayor rango de diseños posible; escalabilidad, para comprobar los límites al incrementar el tamaño de las arquitecturas modeladas; y el balance entre los tiempos de ejecución y la precisión obtenida en las simulaciones.

Esta plataforma de simulación está orientada a modelar tanto sistemas actuales como nuevos diseños de arquitecturas HPC y aplicaciones. De esta forma, dependiendo de los requisitos del usuario, el modelo puede estar enfocado a un conjunto de sistemas, o por el contrario, éste puede estar enfocado en el sistema completo. Por ello, se pueden modelar sistemas distribuidos completos integrando los sistemas básicos en un único modelo, cada uno con su nivel de detalle correspondiente, lo cual proporciona un alto nivel de flexibilidad. Además, esta plataforma proporciona un buen compromiso tanto entre precisión y rendimiento, como en la flexibilidad proporcionada para poder construir un amplio rango de arquitecturas utilizando diferentes configuraciones.

Además, se ha llevado a cabo un proceso de validación de la plataforma de simulación propuesta, comparando los resultados obtenidos en entornos reales con aquellos obtenidos en los modelos análogos. Posteriormente, se han realizado una serie de experimentos para realizar una evaluación y análisis de cómo evolucionan, tanto la escalabilidad como los cuellos de botella, existentes en una arquitectura HPC típica multi-core utilizando diferentes configuraciones. Básicamente estos experimentos consisten en ejecutar 2 modelos de aplicaciones (HPC y checkpointing) en varias arquitecturas.

Finalmente, se han calculado datos de rendimiento de la propia plataforma de simulación con los experimentos realizados. El propósito de este proceso es calcular, tanto el tiempo como la cantidad de memoria necesaria, para ejecutar una simulación concreta dependiendo tanto del tamaño del entorno simulado, como de los recursos disponibles para ejecutar tal simulación.

Abstract

In this thesis we propose a new simulation platform specifically designed for modeling parallel and distributed architectures, which consists on integrating the model of the four basic systems into a single simulation platform. Those systems consist of storage system, memory system, processing system and network system. The main characteristics of this platform are flexibility, to embrace the widest range of possible designs; scalability, to check the limits of extending the architecture designs; and the necessary trade-offs between the execution time and the accuracy obtained.

This simulation platform is aimed to model both existent and new designs of HPC architectures and applications. Then, depending on the user's requirements, the model can be focused on a set of the basic systems, or by the contrary on the complete system. Therefore, a complete distributed system can be modeled by integrating those basic systems in the model, each one with the corresponding level of detail, which provides a high level of flexibility. Moreover, it provides a good compromise between accuracy and performance, and flexibility provided for building a wide range of architectures with different configurations.

A validation process of the proposed simulation platform has been fulfilled by comparing the results obtained in real architectures with those obtained in the analogous simulated environments. Furthermore, in order to evaluate and analyze how evolve both scalability and bottlenecks existent on a typical HPC multi-core architecture using different configurations, a set of experiments have been achieved. Basically those experiments consist on executing the two application models (HPC and checkpointing applications) in several HPC architectures.

Finally, performance results of the simulation itself for executing the corresponding experiments have been achieved. The main purpose of this process is to calculate both the amount of time and memory needed for executing a specific simulation, depending of the size of the environment to be modeled, and the hardware resources available for executing each simulation.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

This chapter describes the scope in which this work has been developed. Moreover, the main contributions of this thesis and the challenges faced to accomplish them are explained in detail. First section 1.1 introduces the definition and the scope of this thesis. Next section 1.2 describes the motivation found and challenges to be addressed in the simulation of High Performance Computing. In the next section 1.3, the main contributions provided by this thesis are explained in detail. Finally, section 1.4 shows the structure of this document is summarized, in order to provide a global idea about what can be found in this document.

1.1 Thesis definition and scope

Nowadays, the development of new computing system networks is a major industry trend, where the design of new and improved computing networks is a very important research field. New deployments have to face high levels of expectations and requirements from the users, which are increasing every day. This level of requirement is expected for everyone and for each component in the system: networks must have a bigger bandwidth, storage management must be bigger and faster, processors must be more powerful, etc. Thus, the whole system must show the performance that all single component promise. But it is difficult to ensure the real capabilities of a new computing system, or an improved one, until it is completely deployed because good components do not ensure a good system.

Furthermore, there is not an issue-free architecture, the system must be perfectly balanced and focused on the task by hand to obtain the best performance. Those difficulties arise because there are so many inter-related parameters that have an important influence in the overall system performance, like the number of nodes that composes the network, the kind of applications that will be executed, communication links characteristics, latencies, etc.

Obtaining the perfect balance is a really hard task. There are a lot of combination and strategies. Moreover, each task or application will require a different balance to get the best results. For selecting and optimizing trade-offs between influential factors it is necessary to perform experiments using as many scenarios as be possible. However, performing this task in real large-scale environments is costly and difficult. To achieve these goals, a feasible solution is to model and simulate the different hardware components and the different application models to obtain and mend the flaws and also to compare performances in

order to achieve the best designs.

Simulation environments have to balance goals like the level of accuracy, the complexity of customization, and the resources required to run them. The level of detail of those simulations can vary greatly from a simple draft up to a detailed representation.

In this work we use the terms: Real Application to refer to the application whose behavior is to be simulated; Real Environment for the hardware-based environment where the Real Application will be executed; Simulated Environment as an environment that represents the features of the Real Environment on a simulation execution; Simulated Application as the behavior of the Real Application running on the Simulated Environment; and Simulator as the program that executes the Simulated Application on a Simulated Environment.

1.2 Motivation

High performance applications have been a huge research field over the past years. The need to obtain better computing architectures, that could reasonably handle new high performance applications of a larger scale than before, is a permanent requirement in the computing research field. Moreover, optimizing this kind of applications for best performance in the design phase is almost impossible due to the complex architecture of such machines.

Major requirements for high performance environments are scalability, reliability and availability. In those environments, defining an architecture that satisfies those requirements is a very difficult and complex task. There are so many inter-related parameters that have an important influence in each one of those requirements, like the number of nodes, the kind of applications that will be executed on the network, communication links, etc. Furthermore, there are tradeoffs in cost, ease of management, and performance directly correlated with those parameters.

Due to this reason, estimating the impact of any design feature or any application on the global system performance becomes of vital importance. Predicting the impact of even small changes on the performance of complex system is a very difficult and non-trivial job. Aspects like detecting system bottlenecks or calculating the scaling degree that some algorithms could obtain, are expensive and time-consuming tasks when performing on a real huge computing network.

Basically, there are two ways to perform studies of parallel and distributed environments. First method is to run the desired application on a real hardware-based system and take measures of the performance obtained. Second method is to run the same application or a simplified version of the application, on a simulated environment that represents the real system. Both methods can be used to analyze, debug and predict the performance of different applications on a variety of architectures. On the one hand, using different architectures with real hardware is a very hard and costly process. On the other hand, developing different implementations for the same HPC application is also a very costly and time-consuming process.

Simulation can simplify those processes at the cost of reducing the accuracy of the results. However, simulation methods have their own advantages and disadvantages. Some of them are:

- Simulation experiments are less expensive and more flexible than hardware-based experiments because they do not require modifying the real system to analyze different possibilities.
- Simulation experiments can be launched on any hardware platform.
- In many cases, simulation experiments are more time-expensive than hardware-based experiments. This problem can be minimized by adding hardware resources for the simulation, for example, parallelizing the simulation execution on a huge computing cluster.
- Results obtained from simulation need to be validated to ensure their accuracy.
- Scaling the architecture of the real system is more expensive and time-consuming that performing the same changes in a simulated environment.
- Simulators can be shared easily with other researchers, while hardware is more difficult to share.

  given to other researchers. While hardware is difficult to transport, a simulator can be sent electronically.
- Simulation only takes care of these aspects we have included on it. Therefore, the possibility that one element not included results to be the key of the performance is always there.

Accurate and efficient performance prediction of large-scale parallel applications on multiple-target architectures is a challenging problem. For those reasons the use of simulation tools should be tempered with the results expected and the state of the design we are involved. A good strategy is to start with soft simulations that cover only few and well recognized aspects of the desired system. This let us to try a great number of combinations and to get results soon. As the correct trend of the design is being obtained the simulations should get more detailed and heavy to run. The number of combinations will be reduced but the results will be more accurate. At the end, the final design should be implanted and experiments should be done in order to get the best configuration possible.

Simulation tools must be prepared to cope with all these changes. In fact, simulation tools should be able to perform simulations with different levels of accuracy, different components, etc. Flexibility is probably the best feature of a good simulation tool. But this flexibility must be easy to use in order to reduce the cost of setting different environments. Finally, accuracy is also a desirable feature. Thus, accuracy should be side by side with flexibility in order to provide faster simulations with lower accuracy, and slower simulations with higher accuracy.

For this reason, simulation tools should keep a balanced ratio between flexibility, scalability and performance, depending of the pursued objectives for which the corresponding simulation tool had been developed for. Ad-hoc simulators are tools that ease the implementation of simulation environments at the cost of reducing the flexibility. In the other side, general simulation frameworks have all the flexibility because researchers can program whatever they want, but the effort of doing so, or making any change is too hard. Those frameworks provide things like communication primitives, predefined modules, statistical

support, configuration mechanisms, etc. which ease the developing. Simulation frameworks specialized on certain domains can deal with the lack of usability. As the framework is more complete and easy to use, the domain that it handles gets reduced. Moreover, speed and accuracy are inversely related. If one of these characteristics increases, the other one decreases.

The challenge that this thesis has to face is two-fold. First, providing a flexible and scalable method to model parallel and distributed architectures, that achieves a good compromise between accuracy and execution speed. Second, providing a method to model and simulate the execution of high performance applications on the previous modeled environments.

1.3 Main objectives

This thesis proposal addresses the challenges previously commented. Its main objective is to **provide new contributions for modeling the execution of high performance computing applications on parallel and distributed architectures**. Thus, for accomplishing this objective we propose the next approaches:

- **Designing strategies for modeling parallel and distributed architectures**. Basically, those strategies have to achieve two main requirements. First, they have to be scalable and parallelizable in order to model large-scale environments, and to execute them in a reasonable amount of time. It also has to achieve a good compromise between accuracy and execution speed for each stage of the design process. Second, those strategies have to be flexible enough for building a wide range of architectures with different configurations. Also, the modeling process has to be faster and easier than deploying and configuring the corresponding real system, which will provide a very important valuable feature for this proposal.

- **Designing strategies for modeling and simulating the execution of high performance computing applications**. Those strategies have to be able for modeling distributed applications to be executed on any modeled distributed environment. Moreover, the proposed strategies for modeling applications should cover from basic and quick drafts to complete ports of the applications, so they can be used in every step of the design process.

- **Analyzing the overall system performance using HPC applications that perform I/O operations masively in different architectures**. Thence, experiments achieved in this case allow to select the architectural configuration that provides the better performance for a given application.

Basically this thesis is focused on evaluating the impact on the system performance by simulating high performance applications on parallel and distributed architectures. Using this approximation we can analyze the behavior of a concrete application in several architectures, without deploying any specific hardware environment. Then, drawbacks and bottlenecks in both studied application and architectures can be located to propose solutions that increase its performance.

1.4 Structure of this document

The rest of this document is organized in two chapters, whose contents are summarized in the following paragraphs:

- Chapter 2, State of the art, is a review of current works about modeling High Performance Computing environments. In this chapter are described most relevant simulators and techniques for modeling all concerning elements to high performance computing, like networks, I/O devices, memories, applications, etc. Moreover, well know high performance computing architectures and high performance computing applications are also described.

- Chapter 3, The SIMCAN simulation platform, shows a proposal of a fast, flexible, scalable and expandable simulation platform for modeling and simulating distributed systems and applications. The main objective of this chapter is to propose a strategy for simulating complex and large environments that represent, both actual and non-existent architectures by modeling individually each one of the four basic systems. Those basic systems consist of computing system, memory system, storage system and network system.

- Chapter 4, Modeling and Simulating computer architectures in SIMCAN, describes the process for modeling environments using the SIMCAN simulation platform. Moreover, some proposed strategies for both ease this process and automatically accomplish the parallelization of those kinds of simulated environments are presented. Finally, the usefulness of those strategies by modeling real environments and showing their performance is presented.

- Chapter 5, Modeling and Simulating applications in SIMCAN, shows several methods and techniques for modeling applications in the SIMCAN simulation platform. In order to predict the performance of any system, it is not enough by modeling and simulating the system architecture. The application to be executed in that environment must be also modeled. However, there are several issues that hamper modeling applications, like data-dependent computation times, inter-process communications and synchronization delays, and other architecture-specific timing information. Due to the great number of existent issues for modeling applications, presented approaches in this chapter try to cover a wide range of applications to be modeled using.

- Chapter 6, Validation of the SIMCAN simulation platform, shows the process for validating the SIMCAN simulation framework. This process consists on modeling several distributed environments and then executing a set of well-known benchmark in both real and simulated environments. Finally, results obtained in both environments are compared in order to check the accuracy of the simulation platform using several configurations and applications.

- Chapter 7, Scalability and Performance experiments, shows experiments for evaluating and analyzing how evolve both scalability and bottlenecks existent on a typical HPC multi-core architecture using different configurations. Moreover, performance results of the simulation itself for executing the corresponding experiments have been achieved. The main purpose of this process is to calculate both the amount of time

and memory needed for executing a specific simulation, depending of the size of the environment to be modeled, and the hardware resources available for executing each simulation.

- Chapter 8, Conclusions and future works, presents the conclusions of this work and also describes some future works.

  Finally, the bibliography used during the elaboration of this work is provided.

Chapter 2

State of the art analysis

This chapter provides an overview of the current state-of-the-art works found in literature. Initially, a description of the main concepts of high performance architectures and high performance applications is described.

Next section describes well-known simulation tools for modeling and simulating high performance architectures. Those simulation tools are grouped in three categories: general purpose simulation frameworks, individual component simulation tools and complete computer architecture simulation tools.

Finally, the most widely used techniques for modeling applications and current performance analysis tools are described in detail.

2.1 High performance computing

High performance computing (in short HPC) covers a wide range of systems, from desktop computers through large parallel processing systems. Almasi and Gottlieb defined a parallel computer as "a collection of processing elements that communicate and cooperate to solve large problems fast". [AG89].

High Performance Computing can be defined as the use of parallel processing for running advanced application programs efficiently, reliably and quickly. The current definition of HPC involved apart from supercomputers, a diverse range of platforms from scalable high end systems to commercial off-the-shelf (COTS) clusters.

The most common users of HPC systems are scientific researchers, engineers and academic institutions. High-performance systems often use custom-made components in addition to so-called commodity components.

2.1.1 High performance architectures

With ever-increasing computing power demands for large-scale applications, the HPC community has been deploying modern computing systems with increasing size and complexity. Basically, those high performance systems can be classified in two models: shared memory and distributed memory.

The shared memory model uses a global address space which can be accessed from

any processor. The high speed for sharing data between processes is the main advantage of this model. Otherwise, the main drawbacks of this model are the lack of scalability and that assuring synchronization and correct accesses to the global memory for all processes is responsibility of the programmer.

The distributed memory model does not use global memory space. Instead, each processor has its own private memory which cannot be accessed from the rest of the processors. Thus, for sharing data the corresponding processors have to exchange messages through a communication network. This model is more scalable than shared memory but is also slower because shared data have to be sent using a communication network.

In recent years, the use of HPC using a clusters are increasing its role and they are very frequently on the TOP 500 list [MSDS10]. The TOP500 list started in 1993 as a project to compile a list of the most powerful supercomputers in the world. It has evolved from a simple ranking system to a major source of information to analyze trends in HPC. A commodity cluster is defined in [Ste01] as a local computing system comprising a set of independent computers and a network interconnecting them.

However, clusters are not a problems-free architecture; each one has advantages and drawbacks. Following, some advantages of cluster systems are listed:

- Cost-effectiveness. High performance clusters are intended to be a cheaper replacement for the more complex/expensive supercomputers.
- Scalability. Cluster computing can scale to very large systems. Hundreds or even thousands of machines can be networked to suit the application needs.
- Availability. Replacing a "faulty node" within a cluster is trivial compared to fixing a faulty SMP component, resulting in a lower mean-time-to-repair (MTTR) for carefully designed cluster configurations.
- Programming model. Programs that use message passing can be easily ported between architectures. Furthermore, shared-memory programming requires that programmers handle the synchronization, which hampers writing applications for this kind of systems.

Parallel processing and parallel computer architectures are a field with decades of experience that clearly demonstrates the critical factors of connection latency and bandwidth, the value of shared memories and the need for lightweight control software. Generally, clusters are known to be weak on all these points. Following, some of those drawbacks of cluster system are listed:

- Communication overhead. Clusters have higher network latency with a lower network bandwidth compared to SMP and supercomputers.
- Reliability. Another potential problem is the frequency of hardware failures (Mean-time-to-failure, in short MTTF). Because of the many heterogeneous commodity hardware involved to build an HPC cluster, the probability of a hardware failure is higher than in SMP machines.

- Coherency. The shared memory model is more closely related to the way applications programmers consider their variable name space. Thus, corresponding hardware can provide more efficient mechanisms for critical functions, as global synchronization and automatic cache coherency.

The complexity of those large-scale systems is growing day by day; which increases the role of benchmarks that measure the performance of those systems, like the High Performance Linpack (HPL) benchmark [DLP03] used in the Top500 list. This benchmark is a collection of FORTRAN sub-routines for solving various systems of linear equations, which are based on a decomposition approach to numerical linear algebra. Another benchmark is High Performance Computing Challenge[LDK+05] (in short HPCC). The HPCC is a suite of tests that examine the performance of HPC architectures using kernels with memory access patterns more challenging than those of the High Performance Linpack.

The HPC Challenge benchmark consists of 23 individual tests that calculate how effectively the system performs high performance computing applications. This benchmark does not measure the theoretical peak performance of a computer, but provide information on the performance of the computer in real applications. The tests do assess criteria that are decisive for the user, such as the rate of transfer of data from the processor to the memory, the speed of communication between two processors within a supercomputer, the response times and data capacity of a network, etc.

Nowadays there are a good number of HPC architectures. One of the most well known and powerful HPC architectures is the Blue Gene/L [MBC+06]. Blue Gene/L (see figure 2.1) has a proven scalability record up to 65,536 dual-processor compute nodes, which includes a variety of nodes with dedicated roles: compute nodes, I/O nodes, service nodes, front-end nodes and file server nodes. Those nodes are interconnected through five networks: a 3D torus network for point-to-point messaging between compute nodes, a global combining/broadcast tree for collective operations over the entire application, a global barrier and interrupt network, a Gigabit Ethernet to JTAG network for machine control, and another Gigabit Ethernet network for connection to other systems, such as hosts and file systems.

For cost and overall system efficiency, compute nodes are not hooked directly up to the Gigabit Ethernet, but rather use the global tree for communicating with their I/O nodes, while the I/O nodes use the Gigabit Ethernet to communicate to other systems. The compute and I/O nodes form the computational core of Blue Gene/L, which are controlled from the service node through an Ethernet control network. The control network is used to control the hardware from the service node. The control system is responsible for operation and monitoring of al compute and I/O nodes. It is also responsible for other hardware components such as link chips, power supplies, and fans.

Compute nodes run an operating system that is dedicated to supporting application processes performing computations. Each Blue Gene/L compute node is a small computer with two processors and its own private memory (each node can support up to 2 GB of local memory) that is not visible to the other nodes. Only compute nodes are interconnected through the torus network. Otherwise, the front-end nodes support program compilation, submission and debugging.

I/O nodes run an operating system that is more flexible and can support various forms of I/O. The file servers store data that the I/O nodes read and write. The I/O nodes plug

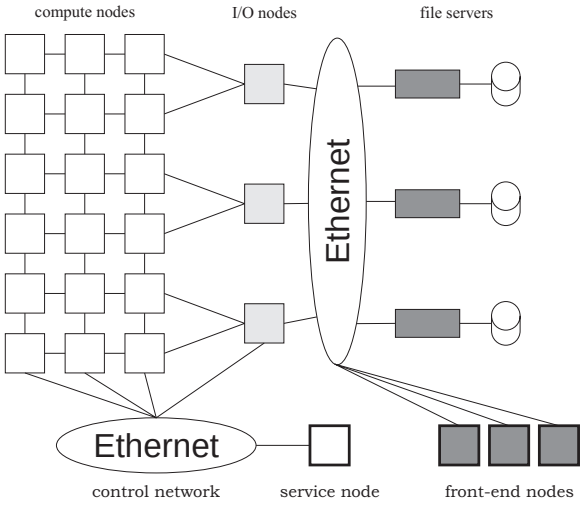


Figure 2.1: Blue Gene/L architecture

into an Ethernet fabric together with the file servers and front-end nodes. The collective and global barrier networks interconnect compute and I/O nodes. The purpose of the I/O nodes during application execution is to complement the compute node partition with services that are not provided by the compute node software. I/O nodes provide an actual file system to the running applications.

Blue Gene/L is a partitionable machine, and can be divided along natural boundaries into electrically isolated partitions. These are 8x8x8 configurations of compute nodes called midplanes. A partition is formed by a rectangular arrangement of midplanes. Each partition can run one and only one job at any given time.

A key concept in the Blue Gene/L operating system solution is the organization of compute and I/O nodes into logical entities called processing sets or psets. A pset consists of one I/O node and a collection of compute nodes. Every system partition, in turn, is organized as a collection of psets. All psets in a partition must have the same number compute nodes, and the psets of a partition must cover all the I/O and compute nodes of the partition. The psets of a partition never overlap. The supported pset sizes are 8, 16, 32, 64 and 128 compute nodes, plus the I/O node.

The psets are a purely logical concept implemented by the Blue Gene/L system software stack. They are built to reflect the topological proximity between I/O and compute nodes, thus improving communication performance within a pset.

The I/O node plays a dual role in Blue Gene/L. On one hand, it acts as an effective master of its corresponding pset. On the other hand, it services requests from compute nodes in that pset. Jobs are launched in a partition by contacting corresponding I/O nodes. Each I/O node is then responsible for loading and starting the execution of the processes in each of the compute nodes of its pset. Once compute processes start running, the I/O nodes wait for requests from those processes. Those requests are mainly I/O operations to be performed against the file systems mounted in the I/O node.

Another well-known HPC supercomputer is MareNostrum. This supercomputer is the most powerful supercomputer in Europe (and the world's fifth most powerful) as of Novem-

ber 2006, according to the LINPACK benchmark. The supercomputer consists of 2560 JS21 blade computing nodes, each with 2 dual-core IBM 64-bit PowerPC 970 processors. It is capable of 62.63 teraflops and a peak performance of 94.21 teraflops according to the LINPACK benchmark. It has 44 racks and occupies only 120 m^2 (less than half a basketball court) and weighs 40,000 kg. Figure 2.2 shows the architecture of MareNostrum.

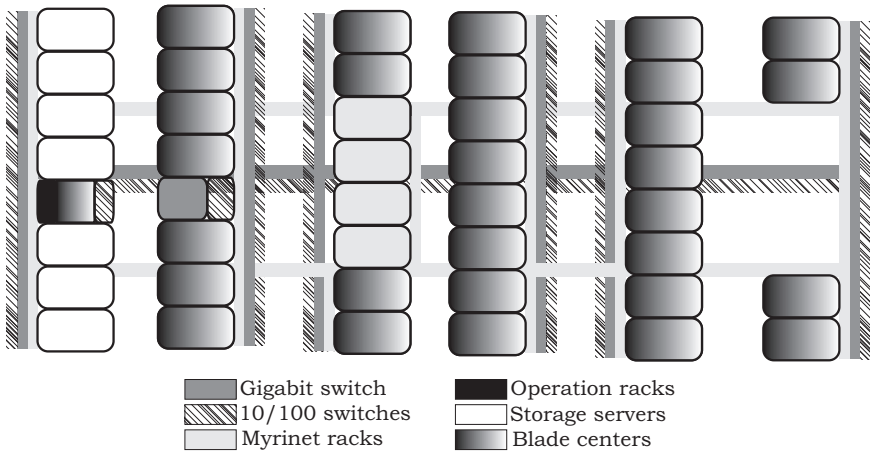


Figure 2.2: MareNostrum architecture

MareNostrum consists of 31 racks dedicated to calculate. These racks have a total of 10240 processors PowerPC 970 with a frequency of 2,3 GHz and 20TB of total memory. Each rack is formed by 6 Blade Centers. In total, each rack has a total of 336 processors and 672 Gb of memory. Each one has a rough peak performance of 3.1 Tflops.

The 2560 blade nodes JS21 are interconnected through a high speed interconnection network called Myrinet. The different nodes are interconnected via fiber optic cables. Four of the 44 racks in MareNostrum are dedicated to network elements which allow interconnecting the different nodes connected to the Myrinet network. These four racks are located in the center of the room and each node has a fiber optic cable. The network elements connect the different cables allowing the interconnection from one point to another from the different nodes.

Further to the local disk of each node with a 36GB capacity, MareNostrum has 20 storage servers arranged in 7 racks. These have a total of 560 disks of 512GB and each one provide a total capacity of 280 TB external storage. These disks are working with GPFS (Global Parallel File System) [GPF09] which offers a global vision of the file system and also allows a parallel access.

The 2560 nodes access the disks through the Gigabit network. Each one of the 20 storage nodes has two nodes p615 in charge of the disk requests, a controller type FAStT100 and one unit EXP100.

One of the racks is the operation rack where the system can be managed. This rack is located in the machine console.

One of the racks of MareNostrum is dedicated to the interconnection of the Gigabit network and one part of the interconnection elements of the Ethernet 10/100 network. It consists of 1 switch Force10 E600 Gigabit Ethernet and 4 Switches Cisco 3550 48-port Fast Ethernet.

2.1.2 High performance applications

Writing a program that uses multiple processors to solve a problem adds several challenges for the programmer, which must be aware of how multiple processors operate together, and how the problem can be efficiently divided among those processors. Depending on the used architecture, the programmer must use a shared memory model or a message passing model. At present, most used models are openMP [CDK+00] and MPI [GHLL+98].

OpenMP is a shared-memory application programming interface (API) whose features, are based on prior efforts to facilitate shared-memory parallel programming. Also, openMP is intended to be suitable for implementation on a broad range of SMP architectures, like multi core machines and multithreading processors.

Like its predecessors, openMP is not a new programming language. Rather, it is notation that can be added to a sequential program in FORTRAN, C, or C++ to describe how the work is to be shared among threads that will execute on different processors or cores and to order accesses to shared data as needed. The appropriate insertion of openMP features into a sequential program will allow many, perhaps most, applications to benefit from shared-memory parallel architectures, often with minimal modification to the code. Figure 2.3 shows the openMP execution model.

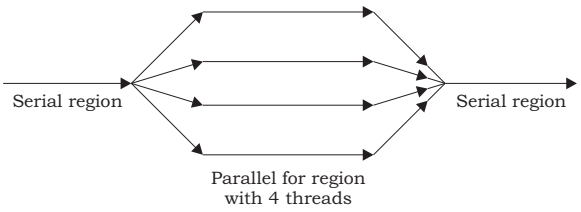


Figure 2.3: openMP execution model

The success of openMP can be attributed to a number of factors. One is its strong emphasis on structured parallel programming. Another is that openMP is comparatively simple to use, since the burden of working out the details of the parallel program is up to the compiler. It has the major advantage of being widely adopted, so that an openMP application will run on many different platforms.

MPI, the Message Passing Interface, is a standardized and portable message passing interface designed by a group of researchers from academia and industry to function on a wide of variety of parallel computers. The standard defines the syntax and semantics of a core of library routines useful to wide range of users writing portable message-passing programs in FORTRAN or C. Figure 2.4 shows the MPI execution model.

Message passing is a programming paradigm used widely on parallel computers, especially Scalable Parallel Computers (SPCs) with distributed memory, and on Network of Workstations (NOWs).

The implementation language for MPI is different in general from the language or languages it seeks to support at runtime. Most MPI implementations are done in a combination of C, C++ and assembly language, and target C, C++, and FORTRAN programmers. However, the implementation language and the end-user language are in principle always decoupled. Some of the most used implementations are:

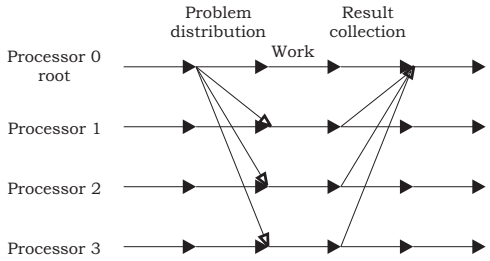


Figure 2.4: MPI execution model

- LAM/MPI [BDV94]: LAM/MPI is a high quality implementation of the Message Passing Interface (MPI) Standard. LAM/MPI provides high performance on a variety of platforms, from small off-the-shelf single CPU clusters to large SMP machines with high speed networks, even in heterogeneous environments. In addition to high performance, LAM provides a number of usability features key to developing large scale MPI applications.
- MPICH [GLDS96]: Currently, MPICH2 [BMG07] is the newest version of MPICH. MPICH2 is an all-new implementation of MPI, designed to support research into high-performance implementations of MPI-1 and MPI-2 functionality. In addition to the features in the previous version of MPICH (MPICH1), MPICH2 includes support for one-side communication, dynamic processes, inter-communicator collective operations, and expanded MPI-IO functionality. Clusters consisting of both single-processor and SMP nodes are supported.
- MPI-LITE [Bha97]: MPI-LITE is a library developed to support multithreaded computation within MPI. With the standard MPI distributions, each process is typically mapped to a unique processor; the only way to map multiple processes to a processor was by creating multiple heavy weight processes. MPI-LITE provides a portable kernel for thread creation, termination, and scheduling. The kernel can be used to spawn multiple light-weight processes (or threads) which execute together as a single process on a processor. Also, MPI-LITE offers significant performance benefits compared to MPI, as the threads mapped to a common processor in MPI-LITE can communicate directly without requiring expensive OS support. Among other benefits, the ability to support multiple threads implies that a programmer can effectively use latency hiding programming techniques to improve parallel program performance.
- Open MPI [GFB$^+$04]: Open MPI is an all-new implementation of the Message Passing Interface. Focusing on production-quality performance, the software implements the full MPI-1.2 and MPI-2 specifications and fully supports concurrent, multi-threaded applications. Its component architecture provides both a stable platform for third-party research as well as enabling the run-time composition of independent software add-ons. Also, Open MPI is capable of both maximizing the achievable bandwidth to applications and providing the ability to dynamically handle the loss of network devices when nodes are equipped with multiple network interfaces. Thus, the handling of network failovers is completely transparent to the application.

Whether running on a parallel computer or not, nearly every program has a mixture

of parts that are serial and parts that are parallel. Gene Amdahl, architect of the IBM 360 computers, developed what is now called the Amdahl's Law to characterize how well an application can make use of scalable parallel processors.

Amdahl's Law [Amd67] looks at how much of the program can be run in parallel and how much of the program must be run using only a single processor. This law is based on fixed workload or fixed problem size. It implies that the sequential part of a program does not change with respect to machine size (i.e, the number of processors). However the parallel part is evenly distributed among n processors. Once the ratio of parallel to serial time is established, it puts an upper bound on the possible speedup for this application using more processors. An argument put forth by Gene Amdahl which establishes that even when the fraction of serial work in a given problem is small, say, s, the maximum speedup obtainable from even an infinite number of parallel processors is only $1/s$. If N is the number of processors, s is the amount of time spent (by a serial processor) on serial parts of a program, and p is the amount of time spent (by a serial processor) on parts of the program that can be done in parallel, then Amdahl's law says that speedup is given by the equation 2.1.

$$speedup = \frac{(s+p)}{(s+p/N)} = \frac{1}{(s+p/N)} \tag{2.1}$$

The total time $s+p=1$ has been established for algebraic simplicity. Figure 2.5 shows the speedup of a program given its sequential fraction using multiple processors.

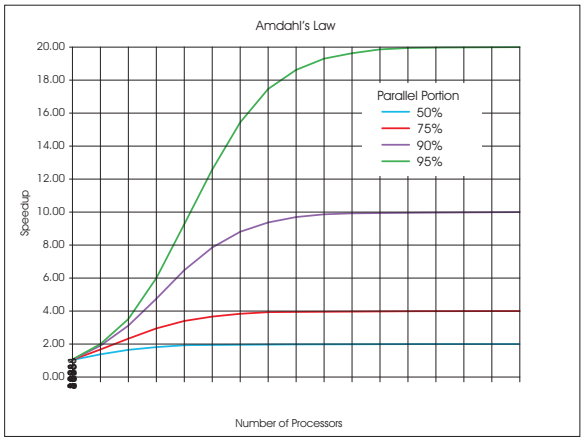


Figure 2.5: Speedup under Amdahls law

Gustafson's Law [Gus88] demonstrate that the assumptions underlying Amdahl's 1967 argument are inappropriate for the current approach to massive ensemble parallelism. Gustafson approaches the problem from another point of view, removing the fixed problem size or fixed computation load on the parallel processors. Instead, he proposed a fixed time concept which leads to scaled speed up. The argument of Gustafson is that the problem size should be increased to meet the available computation power for better results. If P is the number of processors, S is the speedup, and α the non-parallelizable part of the process, Gustafson's law says that the speedup is given by the equation 2.2.

$$S(P) = P - \alpha \cdot (P-1) \tag{2.2}$$

Those theories can be simply explained using the driving metaphor. This metaphor supposes a car that is traveling between two cities 60 miles apart, and has already spent one hour traveling half the distance at 30 mph.

Amdahl's Law approximately suggests "No matter how fast you drive the last half, it is impossible to achieve 90 mph average before reaching the second city. Since it has already taken you 1 hour and you only have a distance of 60 miles total; going infinitely fast you would only achieve 60 mph."

Gustafson's Law approximately states: "Given enough time and distance to travel, the car's average speed can always eventually reach 90mph, no matter how long or how slowly it has already traveled. For example, in the two-cities case this could be achieved by driving at 150 mph for an additional hour."

At present day there are a huge number of HPC applications that cover a wide range of research areas. An example of a HPC application is GROMACS (GROningen MAchine for Chemical Simulations) [BSD95]. GROMACS is a molecular dynamics simulation package originally developed in the University of Groningen, now maintained and extended at different places, including the University of Uppsala, University of Stockholm and the Max Planck Institute for Polymer Research. This work describes a parallel message-passing implementation of a molecular dynamics (MD) program that is useful for bio(macro)molecules in aqueous environment.

Another well known HPC application is SWEEP3D [WHH+00]. SWEEP3D is a particle transport application benchmark designed to be representative of structured grid computations. SWEEP3D maps 3D space onto a 2D grid of processors. Data in the X and Y directions are divided among processors, and data in the Z direction is divided into "K-planes". Data from several K-planes are grouped within a processor to form a block. Blocks are updated starting from a corner and sweeping across the mesh at several different angles. The computation is pipelined across the processor grid for each sweep. A sweep starts in one corner of the grid and proceeds to the opposite corner along the diagonal. When the corner processor has updated its block, it sends its new boundary conditions to its neighbors in the X and Y directions and continues work on the next block. The neighbors process their new data and pass the results on to their downstream neighbors. SWEEP3D's pipelining overlaps communication with computation, reducing execution time when the number of time steps is large.

2.2 Modeling and simulating high performance architectures

Robert E. Shannon defines simulation [Sha98] as *the process of designing a model of a real system and conducting experiments with this model for the purpose of understanding the behavior of the system and/or evaluating various strategies for the operation of the system.* In computer science, simulation is the technique of representing the real world by a computer program, which should imitate the internal processes and not merely the results of the thing being simulated. Simulation models can be classified along three different dimensions:

- Stochastic or deterministic. If a simulation model does not contain any probabilistic component, it is called deterministic. In a deterministic simulation, a system is sim-

ulated under well determined conditions. In this kind of simulations, only one run is needed and there is no truly random variable involved. Furthermore, the output is determined once the set of input quantities and relationships in the model have been specified. Otherwise, stochastic simulation is used when random input values are needed. Stochastic simulation models produce output that is itself random, and must therefore be treated as only an estimate of the true characteristics of the model.

- Continuous or discrete. A continuous simulation uses differential equations (either partial or ordinary), implemented numerically. These types of simulations are most appropriate if the material or information that is being simulated can be described as evolving or moving smoothly and continuously, rather than in infrequent discrete steps or packets. Discrete-event simulation concerns the modeling of a system as it evolves over time, by a representation in which the state variables change instantaneously at separate points in time. These points in time are the ones at which an event occurs.

- Static or dynamic. A static simulation model is a representation of a system at a particular time, or one that may be used to represent a system in which time simply plays no role. On the other hand, a dynamic simulation model represents a system as it evolves over time.

- Local or distributed. A local simulation is executed on a single computer. Distributed models run on a network of interconnected computers. Simulations dispersed across multiple computers are often referred to as "distributed simulations".

At present, it is difficult to ensure the real capabilities of a new computing network, or an improvement over an existing one, until it is completely deployed. For this reason, it is very important to obtain an early accurate estimation of which will be the future performance of the new system. To achieve these goals, the best solution is to use models and simulations of the future computing network.

Currently there is a wide range of simulating tools to study the behavior of computer systems. In next section will be described the most relevant simulation tools, from specific simulators for computer components (like disks or memories) to simulators of large networked systems.

2.2.1 General purpose simulation frameworks

Nowadays, they are toolkits or general purpose frameworks for simulating a wide variety of systems, each one tailored for a specific type of problem. What they all have in common, however, is that they allow the user to model how a system might evolve or change over time. Most of those frameworks are designed as high-level programming languages that allow the user to simulate many different kinds of systems in a flexible way. Following, some of the most free well-known simulation frameworks are listed.

OMNeT++ [Var01] is a C++ discrete event simulator designed for modeling computing networks, parallel systems and distributed systems. OMNeT++ is adequate for large-scale simulations because it uses hierarchical models and reusable components. The basic structure of an OMNeT++ simulation is a set of modules that send and receive messages

among themselves. Those modules send and receive messages across several predefined connections configured between the modules. Those connections conform a network that simulates the desired architecture. Each module has its own value of the local simulated time. This local time only advances when the module receives a new message.

Modules are implemented as C++ objects that inherit from a basic module class. The behavior of the modules can be programmed using two different programming models:

- Event oriented model. Each time a message arrives, a function of the module (handler) is executed, using the message as a parameter of the function.
- Coroutine-based model. The module executes a main function. The messages are processed when the main function executes a *receive* sentence.

The model based on co-routine is more intuitive and easy for developing. But it also requires a great amount of memory to assign a stack to each module. Thus, this model is useless for large-scale simulation. So, the only option for large-scale simulations is the event-oriented model.

The modules have a hierarchical organization that eases the construction of parallel computing architectures. There are two kinds of modules:

- Simple modules, that do not include nothing else that the module itself.
- Compound modules, those include other modules as components. The connections associated to a compound module can be redirected to/from an inner sub-module.

The messages are also implemented as C++ objects that inherit from a basic message class. The messages are essentially a collection of data and some functions. Those functions are used to get/set the data and to perform some treatments to the data. There are several ways to implement a message object:

- Writing an specification of the message data (using .msg format). This specification is pre-compiled to generate the C++ object. This method only works with messages composed by simple values.
- Writing an C++ object that inherits from the one generated with the .msg specification. The new object can modify the .msg specification behavior at will.

Message objects can also contain another message object using the "encapsulate" feature offered by OMNeT (only one message encapsulated per message).

A simulation is configured using a .ned specification. It describes how many instances of each module have to be simulated. It also describes the connections between all these modules. Another advantage is the parameterization of the modules instances using values stored on the specification itself (like the name of each module). This .ned specification is loaded at the beginning of the execution. Thus, the simulation can be completely reconfigured without recompiling the code.

PARSEC [BMT+98] is a discrete event simulation language, which consists of an enhanced C compiler with the capability to define and create simulation entities and constructors for message communication between entities. Basically PARSEC consists of three

primary components: a parallel simulation language called Parsec (parallel simulation environment for complex systems); its GUI, called Pave; and the portable runtime system that implements the simulation algorithms. This simulator has several important drawbacks. The most important one is that the flexibility of this simulator is very poor. Also, large model do not scale well mainly because it uses internally a co-routine system.

DESMO-J [PK05] is an object-oriented framework targeted at programmers developing simulation models. The acronym "DESMO-J" stands for "Discrete-Event Simulation and Modeling in Java". This longer name highlights DESMO-J's two significant properties:

- DESMO-J supports the discrete-event simulation paradigm. In models of this type, all system state changes are supposed to happen at discrete points in time. Between such events the system state is assumed to remain constant. Discrete-event simulation is therefore particularly suitable for systems in which relevant changes of state occur suddenly and irregularly.
- DESMO-J is implemented in Java. Using this framework to build simulation models ultimately results in writing a Java program.

JavaSim [CL99] is a Java implementation of the original C++SIM simulation toolkit [LM93], which supports the discrete-event process-based simulation where each simulation entity can be considered as a separate process. The simulation entities are therefore represented by process objects, which are actually Java objects that possess an independent thread of control associated with them when they are created. These "active objects" then interact with each other through message passing and other simulation primitives in order to realize the operation path of the simulation.

In most cases, a simulation program needs to model the aspects of the real system to correspond to various distribution functions. JavaSim provides random number generators that follow five common distribution functions:

1. Uniform distribution
2. Exponential distribution
3. Erlang distribution
4. Hyper Exponential distribution
5. Normal distribution

These random generators, along with the simulation processes, constitute the core of JavaSim package.

Adevs [MN05] [Nut05] (A Discrete EVent System simulator) is a C++ library for constructing discrete event simulations based on the Parallel DEVS and Dynamic DEVS (dynDEVS) formalisms. DEVS is a formalism for modeling and perform analysis of discrete event systems (DESs); which was invented by Dr. Bernard P. Zeigler [Zei84]. Furthermore, DEVS has been applied to the study of social systems, ecological systems, computer networks and computer architecture, military systems at the tactical and theater levels, and in many other areas. Recent advances in quantized approximations of continuous systems

suggest promising computational techniques for high performance scientific computing (e.g. in the field of computational fluid dynamics).

Also, there are some commercial simulation frameworks. For example, OPNET [Cha99]. This simulator has a commercial license, but it provides a special license for research and students. OPNET provides a convenient tool for hierarchical modeling of a network, including processes (state machines), network topology description, and simulation of different traffic scenarios. However, as noted in [XWHC05], it needs to be adapted for synchronous environments, requiring explicit design of clocking scheme and a distribution network. One of the most important problems of this simulator is the lack of realism of the network protocol stack. Furthermore, IP network architecture and socket interface are missing.

2.2.1.1 General purpose simulation frameworks classification

In this section we provide a brief classification and comparison between the most suitable simulation frameworks for fulfilling the purpose of this thesis (see table 2.1). Due to this thesis is aimed towards high performance architectures, a set of current simulation frameworks that fits with the purpose of this work have been selected. Then, in order to perform this classification, 5 features have been chosen for characterizing each simulation framework.

Framework	License	Community support	Flexible	Scalable	Parallel
OMNeT++	APL	Yes	Yes	Yes	Yes
PARSEC	Free non-profit	No	Yes	Yes	Yes
Desmo-J	GPL	No	Yes	No	No
JavaSim	LGPL	No	Yes	No	No
Adevs	Open source	No	Yes	Yes	Yes
OPNET	Commercial/Research	Yes	Yes	Yes	Yes

Table 2.1: Comparative of simulation frameworks

- License: This feature represents the domain of the software to be used. Currently there are some licenses for software tools. Some of them are:

  Proprietary software licenses. In those licenses, the software publisher grants a license to use one or more copies of software, but that ownership of those copies remains with the software publisher.

  Free software license. This license establishes that the ownership of a particular copy of the software does not remain with the software publisher.

  Open source license, which generally fall under two categories. Those that aim to preserve the freedom and openness of the software itself ('copyleft' licenses), and those that aim to give freedom to the users of that software (permissive licenses). An example of a copyleft Free Software license is the GNU General Public License (GPL). This license is aimed at giving the end-user significant permission, such as permission to redistribute, reverse engineer, or otherwise modify the software.

- Community support: This feature means that the corresponding simulation framework has an active community that supports such framework by providing and main-

taining simulation models. This is probably one of the most important features of a simulation framework aimed to research, because an active community can be very helpful for discussing with other researchers simulation and modeling issues. Furthermore, other models provided by other researchers can be used in the simulation platform for increasing its functionality. Due to the scope of this work, this feature aims to the field of high performance architectures and applications models.

- Flexible: A flexible simulation framework must let users building environments easily, using several component models with different levels of detail. Flexibility also indicates how well the model is structured to simplify modification, allowing design variants or even completely different designs to be modeled smoothly.
- Scalable: A framework can be considered scalable whether the simulator built using such framework are able to simulate large systems, which contains thousands of computing nodes (see table 2.1).
- Parallel: This feature refers to the possibility of performing parallel simulations using the corresponding simulation framework.

First of all, we need a very flexible tool for modeling and simulating a wide range of computer architectures using different configurations. In this case, all those simulation frameworks provide this feature.

Second, the scope of this work is oriented towards modeling and simulating large architectures. Thus, in order to improve performance in the simulation execution, executing simulations in parallel is a required feature. Moreover, due to the great size of such architectures to be modeled and simulated, the corresponding framework must let model and simulate large-scale models.

Third, license plays a very important role in this study. At this point, two simulation frameworks fulfill the required features. In one hand OMNeT++ has a very active community and a GPL license, which means that both the entire framework and the simulation models provided by other researchers can be used for free. Currently there are a wide range of models to be used with OMNeT++, like INET [Var07]. In the other hand, OPNET also has an active community but its license is not totally free. In this case, there are two kinds of licenses. The commercial license provides full access to the simulation framework, but the research license provides a limited access of such framework. Moreover, this license must be renewed every year, and the work developed must be kept in an updated website.

Therefore, OMNeT++ is the best option due to its APL (Academic Public License) license, the very active community, and the great variety of simulation models provided by its community, which extends the functionality of this framework making it very suitable for the scope of this thesis.

2.2.2 Simulation of individual computing components

Many special purpose simulators exist to simulate very specific types or parts of systems. Those simulators are highly specialized to solve a particular type of problem. In most cases, those simulators require great expertise. In this section will be commented several simulators and techniques, which are in charge of modeling the behavior of specific parts of a complete system, like networks, I/O subsystem elements, etc.

2.2.2.1 Network simulators

Network simulators are focused on a wide range of topics. Many target a specific area of research interest such as a particular network type or protocol. Others target a wider range of protocols. The multi-protocol network simulators can provide a rich environment for experimentation at low cost. Next, several network simulator with their most relevant features will be described.

Cnet [McD91] is a network simulator which enables experimentation with various data-link layer, network layer, routing and transport layer networking protocols in networks consisting of any combination of point-to-point links and IEEE 802.3 Ethernet segments. With reference to the OSI/ISO Networking Reference Model, cnet provides the application and physical layers. User-written protocols are required to "fill-in" any necessary internal layers and, in particular, to overcome the corrupted and lost frames that cnet's physical layer randomly introduces. In addition, advanced users may develop different application and physical layers which exhibit varying statistical characteristics of message generation and data transmission.

The INET framework [Var07] is an open-source network simulation package built upon OMNeT++ that uses the same concept: modules communicating by message passing. This framework contains IPv4, IPv6, TCP, UDP protocol implementations. Supported link-layer models are PPP, Ethernet and 802.11.

Basically, protocols are represented by simple modules where those modules can be used in hosts and other network devices to model the behavior of a real distributed environment. The INET framework also contains several examples of pre-assembled modules like host, router, switch, access point, etc.

NS-2 [NS2] [HRFR06] is a discrete event simulator targeted at networking research. This simulator provides substantial support for simulation of TCP, routing, and multicast protocols over wired and wireless (local and satellite) networks. NS-2 was first released in 1996. It derives from earlier work on S. Keshav's REAL Simulator [Kes88] and the original NS [BBE+99] (NS-1) simulator released by Lawrence Berkeley National Laboratory in 1995. NS-2 is a major architectural change from NS-1, the simulator became entirely based on the blend of MIT Object Tcl (OTcl) and C++.

The core of NS-2 is written in C++, but the C++ simulation objects are also linked to shadow objects in OTcl. Simulation scripts are written in the OTcl language (an extension of the Tcl scripting language). This structure permits simulations to be written and modified in an interpreted environment without having to resort to recompiling the Simulator each time a structural change is made. In the timeframe that NS-2 was introduced (mid-1990s), this provided both a significant convenience in avoiding many time-consuming recompilations, and also allowing potentially easier scripting syntax for describing simulations. NS-2 has a companion animation object known as the Network Animator (NAM), used for visualization of the simulation output and for (limited) graphical configuration of simulation scenarios. Presently, NS-2 consists of over 300,000 lines of source code, with probable a comparable amount of contributed code that is not integrated directly into the main distribution.

Next version of NS (called NS-3) [HLR08] is not an extension of NS-2; it is a new simulator. The two simulators are both written in C++ but NS-3 is a new simulator that does not support the NS-2 APIs. Some models from NS-2 have already been ported from

NS-2 to NS-3. The main goal of the NS-3 project is to produce a discrete event network simulator for Internet systems, with an emphasis on layers 2-4 of the network stack, targeted primarily for research and educational use.

2.2.2.2 I/O subsystem simulators

The complexity of large-scale systems entails the need of storing and managing huge amounts of data efficiently. In a large network, the most common way to store data is to use a fast subnet that establishes a direct connection between storage devices and nodes that request data stored on those devices. Storage subsystem performance is one of the major concerns that arise on this kind of large computing networks. The I/O system is usually a system bottleneck in most computing systems [PGK88]. Due to the impact of the I/O subsystem on the overall system performance, modeling and simulating I/O elements like disks and file systems, become of vital importance.

There are a lot of works that study service disk time simulations and how to calculate what are the most important features involved on it. The most common technique is using the disk physical parameters to simulate a disk (heads, cylinders, tracks, ...) [RW94]. Detailed disk simulators with accurate timing models have been produced in the past, like [WAA+04].

A very well known disk simulator is DiskSim [BSSG08]. DiskSim is an efficient, accurate and highly-configurable disk system simulator developed to support research into various aspects of storage subsystem architecture. This simulator includes modules that simulate disks, intermediate controllers, buses, device drivers, request schedulers, disk block caches, and disk array data organizations. In particular, the disk drive module simulates modern disk drives in great detail and has been carefully validated against several production disks. Furthermore, DiskSim by itself simulates only the performance-related aspects of the storage subsystem. It does not model the behavior of the other computer system components or interactions between them and the storage subsystem.

To simulate a disk accurately and precisely, a huge list of parameters must be configured. This task is at best tedious. There are some works that automatically collect disk drive characterizations, like DIXtrac [SG99]. DIXtrac is a program that automatically characterizes the performance of modern disk drives. As a simulator gets more detailed and takes into account more disk parameters, its performance more closely approximates the performance of the real disk drive. Without human intervention, DIXtrac can discover accurate values for over 100 performance-critical parameters. Furthermore, DIXtrac complements detailed simulators like DiskSim in that it automatically extracts the necessary parameters from a SCSI disk drive, allowing them to be fed into DiskSim for later system simulation and/or testing.

[DSSP06] describes the design and implementation of the Vesper disk drive simulator. Vesper is an instructional disk drive simulator with a high degree of performance realism. This simulator retains simplicity while providing timing statistics close to that of real disk drives. The key to this approach is to provide hardware abstractions that are simple but yet capable of capturing device interactions with major performance impacts. Unlike detailed disk simulators such as DiskSim, Vesper does not differentiate the effects which the controller, bus, and drive have upon the aggregate timing of the whole subsystem. Instead, Vesper views the controller, bus, and drive as a black box which accepts a simple

set of commands. The Vesper profiler accumulates timing statistics for these commands with varying input parameters and records them into a profile. All the characteristics and quirks of the profiled drive can then be incorporated into the Vesper disk simulator.

Also there are works that use analytical models to represent the behavior of disk drives. An example of this kind of modeling is [SMW98]. This work presents an analytic model for disk drives that do read ahead and request reordering. The authors of this work model complex storage devices by modeling the individual physical components of the device, such as queues, caches, and disk mechanisms, and then composing the components to give a composite device model for the entire storage device. Each component model takes as input a workload characterization, transforms this to a workload to impose on lower-level components. In turn, the performance predictions of a component typically depend on the service time predictions of the component(s) on which it relies.

Other component that has an important impact on the overall I/O subsystem performance is the file system. Currently, file systems are a research subject in the I/O field; there are numerous proposals in order to improve its performance, scalability and other features [CCC+03] [CIRT00]. An example of a file system simulator is shown in [She06], but it has a low detail level because it is very simple and for educational purposes. There are other works like [Tan01] [Nol07] which are based on the emulation of a file system.

There are also proposals to simulate file systems [TWL94] [SM06] [Hac92]. Some of them are very detailed (using real metadata for managing file distribution). This approach is quite complex because it requires to implement a simplified prototype of the real file system that has to be changed completely with each file system implementation.

Thus, the main trend in this topic is to obtain statistical estimations for one or several features of a corresponding file system. There are several works [Vog99] [DB99] [Mit03] focused on estimating the file size distribution. Those studies are normally used to producing benchmarks (like the ones used for benchmarking Web Servers). In contrast, there are few works focused on other file system features, specially the way file blocks are distributed across the disk. Obtaining the block distribution of a certain file across the disk is important because the access time of a certain block on the disk is not uniform. In fact, the disk access time depends on the order that blocks are requested.

There are also some efforts for making good file system benchmarks and traces [HYZ05] [BDK97] that can mimic the real behavior (like the aging of a real file system [SS97]) in order to obtain an accurate simulation of the file system.

Due to real storage systems are complex, sometimes more sophisticated models to represent the behavior of those systems are needed. For instance, most storage servers use RAID systems instead of single disk drives, which are more difficult to model and simulate. This work [DBP+04] describes Shear, a user-level software tool that characterizes RAID storage arrays. Shear employs a set of controlled algorithms combined with statistical techniques to automatically determine the important properties of a RAID system, including the number of disks, chunk size, level of redundancy and layout scheme.

2.2.2.3 Memory simulation

Memory system performance is sensitive to a large number of parameters. Each one of these parameters takes on a number of values and interacts in fashions that make overall

trends difficult to discern. At present, there is a lack of tools in the public-domain that support studies related to memory system performance.

The work [WGT+05] describes DRAMsim, a simulator that implements detailed timing models for a variety of existing memories, including SDRAM, DDR, DDR2, DRDRAM and FB-DIMM, with the capability to easily vary their parameters. It also models the power consumption of SDRAM and its derivatives. Furthermore, DRAMSim can be used as a standalone simulator or as part of a more comprehensive system-level model. The authors of this work have successfully integrated DRAMsim into a variety of simulators including MASE [LC01], BOCHS [Boc05] and GEMS [MSB+05].

Another work that model the behavior of the memory system is [MW95], which describes a technique used for efficient simulation of memory in SIMICS [MCE+02]. Its design has focused on efficiently supporting the simulation of multiprocessors, analyzing complex memory y hierarchies and running large binaries with a mixture of system-level and user-level code. A well-defined internal interface to generic memory simulation simplifies user extensions. Leveraging on a flexible interpreter based on threaded code allows runtime selection of statistics gathering, memory profiling, and cache simulation with low overhead.

To evaluate several architectures, some type of high-level simulation is required, including high-level cache simulation. In addition, cache simulation is inevitable to obtain a good performance approximation for modern processors, which are used in computer systems to reduce average memory access times. Existing techniques for predicting cache performance are often unsatisfactory in terms of cost or performance because cache structures are especially difficult to simulate at a high level, with few current tools to accurately simulate cache structures above the instruction set level.

The work [PMP+04] tackle this problem by developing a method for simulating cache structures at a high level quickly and accurately. It proposes to use an ISS (Instruction Set Simulator) or designer intuition to generate short, concise metrics that describe the memory behavior of individual program fragments in the concurrent application set. These metrics are annotated into the original source code, and then this source code and the hardware architecture are executed in the MESH high level performance simulation framework. Since the cache behavior for each program fragment is represented, the cache structure can be an input into the simulator for each processor in the system. The simulator can then find the cache behavior for each processor when executing each application fragment. Using this information, it can determine the interactions between concurrently executing software and discover the net performance of the heterogeneous processor system.

Another work that simulates a cache system is [DJS89]. The authors of this work present a method for efficiently simulating the effects of a cache on the execution time of a program. They use an execution-driven simulation approach that requires no hardware support and provides a highly accurate dynamic address trace to a cache simulation model. Almost all of the overhead in this approach is in the cache simulation rather than the address trace generation. The cache simulator is used in conjunction with the Rice Parallel Processing Testbed [CMMS88] to study the performance of concurrent programs executing on multiprocessor systems with caches. The authors also have been developed an estimative execution-driven simulator that greatly reduces the simulation overhead by using parameters extracted from a detailed simulation of a program's execution on a processor with a cache, along with an analytical model of cache behavior. The predictions

and overhead of the estimative technique are compared with those obtained from detailed cache simulations.

Cache Miss Equations [GMM99] are another technique to represent the cache access patterns of software. Potential cache misses are represented as a set of linear equations whose solutions describe exactly when and where misses occur. When applied to this problem, their inability to handle data-dependent computation and the large amount of data needed prevent them from being widely used.

SMPCache [RPP01] is a simulator for cache memory systems on symmetric multiprocessors. The simulator has been conceived as a tool for the teaching of cache memories on multiprocessors systems. This tool is very useful to evaluate and understand different design alternatives: the number of processors, the cache coherence protocols, schemes for bus arbitration, mapping, replacement policies, cache size, memory block size, etc.

2.2.2.4 CPU simulation

Nowadays, processor simulators are indispensable for both hardware and software system architects to verify and/or to evaluate the functionality and performance of their system.

RealView Development Suite (RVDS) [arm08] is a toolset for building, debugging, and managing software development projects targeting ARM architecture-based processors. RVDS provides a coordinated development environment for embedded systems applications running on the ARM family of RISC processors.

The work [NTN06] proposes a simple but efficient technique for Instruction Set Simulators (ISS). The simulator presented in this work is made workload specific by a simple process to generate a set of C functions from a binary workload. It is as portable and re-targetable as ordinary instruction emulators because the translation targets C code and works well with well-abstracted instruction definitions. The translation is also easy-to-implement without requiring any complicated analysis nor profiling.

FastSim [SL98] is a direct-execution simulator of a speculative, out-of-order uniprocessor with non-blocking caches. Its two primary contributions are speculative direct-execution, which efficiently performs the functional simulation of a program, and fast-forwarding, which dramatically accelerates the time-consuming simulation of an out-of-order micro-architecture. FastSim allows mispredicted branch paths to be executed directly, and then rolled back. Without further optimization, FastSim runs 1.1-2.1 times faster than the well-known SimpleScalar [ALE02] out-of-order simulator, which does not use direct-execution. FastSim's primary contribution is the application of *memoization* result caching to the expensive process of simulating an out-of-order micro-architecture. Traditionally, *memoization* was used to implement functional programming languages by caching function return values. Expensive computation can be avoided by returning a previously cached value, when available.

Another CPU simulator is Shade [CK94], which combines efficient instruction-set simulation with a flexible, extensible trace generation capability. In fact, Shade performs cross-architecture simulation. Efficiency is achieved by dynamically compiling and caching code to simulate and trace the application program. The user may control the extent of tracing in a variety of ways; arbitrarily detailed application state information may be collected during the simulation, but tracing less translates directly into greater efficiency.

A simulator of a relatively modern processor as the Cell processor [KDH+05] is CellSim [CRR+07b] [CRR+07a]. CellSim is a modular simulator for heterogeneous multiprocessors. Its modularity is based on the fact that it is composed by modules, which are connected among them through signals. The authors of these works have used UNISIM [ACG+07] as the supporting infrastructure for module description and connection handling. UNISIM requires that modules are C++ classes with methods sensitive to the modules communication signals, and it is UNISIM who takes care of calling the methods and synchronize the modules. The UNISIM version used requires a clock signal, common to all modules, through which the simulator executes the functionality of modules each cycle. The modules corresponding to the current version of the Cell Broadband Engine Architecture have been implemented, the Cell Processor. CellSim has been configured using these modules in order to be able to validate it against the real processor.

2.2.3 Complete computer architecture simulation

Research studies of large-scale systems has forced simulation tool developers to expand the scope of their simulation tools, both for modeling system components beyond the processor and memory hierarchy, and for supporting execution of unmodified operating systems with commercial workloads for which source code is unavailable.

Currently there are a wide range of simulators. Some of them are focused on simulating with a very high level of detail the entire system by providing functional execution of unmodified commercial operating systems and applications. Those simulators are called also full-system simulators.

The defining property of full-system simulation compared to an instruction set simulator is that the model allows real device drivers and operating systems to be run, not just single programs. Thus, full-system simulation makes it possible to simulate individual computers and networked computer nodes with all their software, from network device drivers to operating systems, network stacks, middleware, servers, and application programs. The main advantage of those simulators is the high level of accuracy obtained, and the main drawback is its performance, which is five or six orders of magnitude slower than a real system.

Early work in academia included the PDP-11 emulator [DM84] developed by John Doyle and Ken Mandelberg and the implementation of g88 [Bed90] by Robert Bedichek. The g88 implementation was subsequently placed in the public domain, and the design details were published. This implementation modeled a single processor M88100-based system with a mixture of real and pseudo devices, which could boot an operating system (specifically, Unix). A predecessor of Simics [MCE+02], gsim [Mag93], begun in 1991, was based on g88 and extended to include support for multiple processors with shared physical memory. In 1994, the gsim simulator was rewritten as a multiprocessor Sparc V8 model, resulting in the first version of Simics.

COTSon [AFF+09] is a simulator framework jointly developed by HP Labs and AMD. The goal of COTSon is to provide fast and accurate evaluation of current and future computing. It targets cluster-level systems composed of hundreds of commodity multicore nodes and their associated devices connected through a standard communication network. COTSon adopts a functional-directed philosophy, where fast functional emulators and timing models cooperate to improve the simulation accuracy at a speed sufficient

to simulate the full stack of applications, middleware and OSs. COTSon's approach to simulating a cluster is radically different from previous approaches for simulating parallel machines. COTSon combines individual node simulators to form a cluster simulator.

Simics [MCE+02] is a platform for full-system simulation, which attempts to strike a balance between accuracy and performance. Simics was one of the first academic projects in this area and the first commercial full-system simulator. Also, this platform is sufficiently generic to model embedded systems, desktop or set-top boxes, telecom switches, multiprocessor systems, clusters, and networks of all these items.

The simulation of processors in Simics is performed at the instruction-set level, including the full supervisor state. Currently, Simics supports models for UltraSparc, Alpha, x86, x86-64 (Hammer), PowerPC, IPF (Itanium), MIPS, and ARM. Simics views each target machine as a node, representing a resource such as a Web server, a database engine, a router, or a client. A single Simics instance can simulate one or more nodes of the same basic architecture, where heterogeneous nodes can be connected into a network controlled by a tool called Simics Central. In addition, Simics facilitates the inclusion of approximate cache and I/O timing models, allowing a first-order approximation of the interleaving of memory operations for a next generation system.

Another full-system simulator similar to Simics is SimOS. SimOS [RHWG95] [RBDH97] is an environment for studying the hardware and software of computer systems. SimOS simulates the hardware of a computer system with enough detail to boot a commercial operating system and run realistic workloads on top of it. By selecting the appropriate combination of simulation models, the user can explicitly control the tradeoffs between simulation speed and simulation detail. Although the SimOS direct-execution mode runs the target operating system and applications quickly, it does not model any aspect of the simulated system's timing and may be inappropriate for many studies. Furthermore, it requires compatibility between the host platform and the architecture under investigation.

To support more detailed performance evaluation, SimOS provides a hierarchy of models that simulate the CPU and MMU via software for more accurate modeling of the target machine's CPU and timing. Moreover, SimOS simulates a large collection of devices supporting the target operating system. These devices include a console, magnetic disks, Ethernet interfaces, periodic interrupt timers, and an inter-processor interrupt controller. This simulator also supports interrupts and direct memory access (DMA) from devices, as well as memory-mapped I/O (a method of communicating with devices by using loads and stores to special addresses).

M5 [BDH+06] is a full-system simulator that has been developed specifically to enable research in TCP/IP networking, which supports the execution of the entire system, including operating system code and models of network and disk devices. M5 is implemented using two object-oriented languages: Python for high-level object configuration and simulation scripting and C++ for low-level object implementation. Furthermore, M5 includes a variety of object models implemented upon the core simulation engine. These models include CPUs, caches, buses, and I/O devices; everything necessary for modeling networks of complete systems. The main disadvantage of this simulator is that it is very slow. For example, a simulation of a two-core system running the Apache server could obtain a slowdown of about 53000:1.

Bochs [Boc05] is a program that simulates a complete Intel x86 computer. It can be

configured to act like a 386, 486, Pentium, Pentium II, Pentium III, Pentium 4 or even like x86-64 CPU, including optional MMX, SSEx and 3DNow! instructions. Also, Bochs interprets every instruction from power-up to reboot, and has device models for all of the standard PC peripherals: keyboard, mouse, VGA card/monitor, disks, timer chips, network card, etc. Because Bochs simulates the whole PC environment, the software running in the simulation "believes" it is running on a real machine. A remarkable feature of this simulator is that it is geared towards virtualization rather than hardware exploration.

A well-known and popular architecture simulator in the simulation field is SimpleScalar [ALE02]. SimpleScalar is a toolset that provides an infrastructure for simulation and architectural modeling. The toolset can model a variety of platforms ranging from simple unpipelined processors to detailed dynamically scheduled micro-architectures with multiple-level memory hierarchies. In addition, SimpleScalar includes instruction interpreters for the ARM, x86, PPC, and Alpha instruction sets. The interpreters are written in a target definition language that provides a comprehensive mechanism for describing how instructions modify registers and memory state. The I/O emulation module provides simulated programs with access to external input and output facilities. SimpleScalar supports several I/O emulation modules, ranging from system-call emulation to full-system simulation.

TFsim [MHW02] is a full-system multiprocessor performance simulator which models a pipelined, out-of-order micro-architecture in detail and the memory system. The main contribution of this work is the definition of a timing-first decoupled simulation approach, in which the timing simulator executes each dynamic instruction ahead of a functional simulator, Simics [MCE+02] in this case. The timing simulator models micro-architectural features with enough detail to model speculative execution and predict the interleaving of inter-thread events. To do this, the timing simulator must also model architectural function mostly correctly. When the timing simulator commits instructions, it invokes the functional simulator to verify if the timing simulator has deviated from the functional simulator. On a deviation, the timing simulator's state is repaired to guarantee functional fidelity. Timing-simulation can be viewed as an almost correct integrated simulator followed by a correct functional simulator checker.

The main difference between a functional simulator and a timing simulator is that, while the first one uses a functional component to produce a logical stream of committed instructions that are fed to a timing component, the second one directs a functional simulator to execute speculative paths and to select thread interleaving.

The timing simulator in this work does not model devices. As such, it cannot model device accesses or interrupts. However, the timing simulator is able to detect when these events occur and correctly model their timing. Device accesses cannot be speculatively executed in real machines, as they have side-effects that can be non-recoverable. When the timing simulator detects that an instruction is accessing a device (as its physical address is in the I/O range), it delays producing a value until retirement. At retirement, it copies the value from the functional into the timing simulator, imitating the non-speculative execution of a real system. Interrupts are similarly detected and handled at retirement time.

UNISIM [ACG+07] is a modular simulation environment, implemented as a layer on top of the industry standard SystemC [sys03]. Besides modularity, a key contribution of the UNISIM environment is a particular focus on the reuse of control logic, which corresponds to a large share of simulator code, and which is often overlooked by simulation

environments. This simulation environment supports an abstract level of modeling, called Transaction-Level Modeling (TLM), in addition to the more common detailed Cycle-Level Modeling (CLM). TLM simulators are less accurate but much faster than CLM simulators. UNISIM allows hybrid CLM/TLM simulators which can zoom in on only the important architecture details. For full-system simulations, UNISIM provide several simulators capable of booting a complex operating system like Linux. These functional simulators can be plugged into CLM or TLM simulators which are compliant to a functional simulator API. Several groups are developing models which will be included in the library: the full-system PowerMac G3/G4 was jointly developed by CEA and BSC, a cycle-level model of the IBM Cell was developed at UPC/BSC [CRR+07b], a model of an ST231 VLIW processor, and later on a distributed-memory multi-core, is being developed at INRIA, an ARM9 cycle-level and full-system simulator is being developed at CEA.

The major drawback of using full-system simulators to model and simulate the behavior of system architectures is the slow-down execution factor. To mitigate those prohibitively slow simulations, researchers often use abbreviated instruction execution streams of benchmarks as representative workloads in design studies. This technique is called sampling [SPHC02] [WWFH05], which is a general statistical technique used in experiments with a large data set to obtain a smaller representative set. In trace sampling, a program trace length is reduced by periodically sampling the program execution. The smaller sampled trace is then used as input for model simulation. Some researchers predominantly skip the initial 250 million to two billion instructions and then measure a single section of 100 million to one billion instructions. However, this technique rarely captures representative behavior.

SimFlex [HSW+04] is a component-based framework for creating timing models of single processor and multiprocessor server systems running commercial applications. This simulation framework uses component-based design and rigorous statistical sampling to enable development of complex models. SimFlex leverages the technology of the commercially-available Simics simulation tool [MCE+02] to provide functional execution of unmodified commercial operating systems and applications. SimFlex provides a framework for rapidly building timing models which augment the system emulation performed by Simics. This simulator applies the SMARTS methodology [WWFH03b] for choosing and rapidly measuring a representative sample of each workload. SimFlex extends SMARTS to multiprocessor simulations, and provides support for the development of the code for warming model state that is essential to achieving unbiased measurement with SMARTS.

Gems [MSB+05] is a simulation toolset to characterize and evaluate the performance of multiprocessor hardware systems, commonly used as database and web servers. The authors of this work leverages an existing full-system simulator (Simics) as the basis around which to build a set of timing simulator modules for modeling the timing of the memory system and microprocessors. Gems has been designed as a modular simulation infrastructure that decouples simulation functionality and timing. In order to obtain both the efficiency and the robustness of a functional simulator the functionality and timing simulation in GEMS have been decoupled. Using modular design provides the flexibility to simulate various system components in different levels of detail. Also, this simulator infrastructure enables running architectural experiments using a suite of scaled-down commercial workloads.

Both SimFlex and Gems have detailed multiprocessor memory systems but lack detailed I/O models and multiple-system capability. In addition, they both rely on Simics to

provide much of the I/O and privileged-mode modeling. This approach reduces development effort, but the resulting black-box nature of the functional model restricts flexibility.

Rsim [HPRA02] is an execution-driven simulator for simulating shared-memory multiprocessors (and uniprocessors) built from processors that aggressively exploit instruction-level parallelism (ILP). Rsim provides the user with a number of configuration parameters to simulate a variety of shared-memory multiprocessor and single processor configurations. For remote communication, Rsim supports a two dimensional wormhole-routed mesh network. For deadlock avoidance, the system includes separate request and reply networks. Also, Rsim simulates applications compiled and linked for SPARC V9/Solaris using ordinary SPARC compilers and linkers.

Talisman [Bed95] is a simulator that models the execution semantics and timing of a multicomputer. Talisman models the semantics of virtual memory, a circuit-switched inter-node interconnect, I/O devices, and instruction execution in both user and supervisor modes. It also models the timing of processor pipelines, caches, local memory buses, and a circuit-switched interconnect. This simulator executes the same program binary images as a hardware prototype at a cost of about 100 host instructions per simulated instruction. Thus, Talisman translates instructions to threaded code, which is then executed. The threaded code is cached, so that the price of translation for most instructions is paid just once, the first time they are encountered in the code stream. The result is a simulator that has a slow-down of about 100 per simulated processor.

EPG-sim [PY93] is a set of general-purpose execution-driven tools that performs parallel simulation and trace generation for studying parallel systems. These tools can model varying processor and system architectures, and can simulate the effects of serial, optimistically parallelized, or parallel codes being executed on modeled parallel systems. EPG-sim allows critical path simulation (CPS), execution-driven trace generation (ETG), and execution-driven simulation (EDS) to be driven by serial, optimistically parallelized, or parallel codes. This can be done because of the extension of CPS instrumentation techniques to ETG and EDS. EPG-sim allows instrumented parallel codes to execute on single processor or parallel hosts, and allows optimistically parallelized or parallel codes to drive parallel simulations. CPS cannot easily model memory latency effects caused by memory contention, network contention, or cache coherence activity. Constant memory delays are usually assumed; however instrumentation can distinguish between intra-task and inter-task memory accesses and assign different delays accordingly.

The SimUTC toolkit [WGSS99] is a fault-tolerant distributed systems simulation built upon the discrete event simulation package C++SIM [LM94]. The SimUTC toolkit provides a flexible environment for both simulation and experimental evaluation of round-based clock synchronization algorithms in distributed systems. Due to its layered, modular and distributed architecture, SimUTC is a powerful instrument for evaluating the performance of such algorithms. SimUTC provides elaborate simulation models for both network and clocks. In fact, when aiming at high accuracy clock synchronization in the 1 μs range, many system parameters ranging from clock granularities up to oscillator stability must be taken into account and, hence, appropriately modeled. Various fault-injection capabilities in conjunction with customized data analysis features facilitate in-depth simulation studies over long periods. A key issue in the design of SimUTC has been the full compatibility with hardware-based simulation. Therefore, it is possible to replace the simulated network and clock modules by real network controllers and clock devices with minimal changes.

A forthcoming paper will be devoted to this framework for comprehensive experimental evaluation. All SimUTC functions are controlled via a user-friendly GUI, which supports system configuration, simulation control, and data analysis.

SimSANs [Zhu09], Simulating Storage Area Networks, is a Data Center Storage Networking design and simulation tool. It is especially useful in infrastructure design and performance analysis of modern Fiber Channel (and FCoE) based data center storage networks. The current version of SimSANs is version 3 and it only runs on Windows platforms. Current SimSANs v3 Engine binary is built with APIs from OMNeT++ [Var01]. The modules cannot be modified due to the source code is not provided, only the precompiled libraries. This software tool includes three components:

- Backend Simulation Engine: the simulation core, designed in C++ and based on top of OMNeT++ discrete event simulation framework.
- Backend Management Agent: the management core, designed in C# and used to control and operate the simulation engines by accepting the requests from management console.
- Frontend Management Console: the GUI, designed in C# and used as a sole management interface for users to remotely control and operate multiple backend simulation engines, including configure, launch, and monitor simulations.

There are also other works focused on distributed storage architectures. One example of this kind of system is MIDAS (Modeling Infrastructure for Dynamic Active Storage) [TSG08]. MIDAS is an execution-driven simulator that captures both the processing and I/O behavior of active storage systems. MIDAS simulates a host system interacting with the I/O path via an interconnection network. The simulated I/O path can include disk drives with programmable processors and programmable storage controllers. The micro-architecture of each one of these components is configurable. Using this framework, the effects of different processor micro-architectures, physical disk and network designs, and communication protocols on application performance can be explored.

The basic building block in MIDAS is the Processing Element (PE) model. The PE model consists of a processor, which is capable of running ad-hoc code, interacting with a disk drive. The processor and disk models are glued together via a layer called the Space Manager. The starting points for building MIDAS are Simplescalar [ALE02] and Disksim [BSSG08], which simulate the processor and disk respectively.

2.2.4 Simulators classification

Currently, due to the high number of simulation tools and the different characteristics of each one of them, making a common list of features that allow classify those tools becomes a very difficult task.

Those features will depend highly of the researcher's objectives. In some cases, designers optimize a model for performance and detail at the expense of flexibility. Designers typically employ these models when they need to faithfully represent a device at speeds capable of executing large workloads, but don't need to change the model.

Developing a universal simulator that satisfies the requirements of all researchers is impractical and unfeasible. Each simulator has its own features imposed for the own simulator's design. Thus, each researcher will choose the simulator that fits the most with the objectives he/she pursue. Generally, when a simulator has a remarkable feature, it entails tradeoffs with other ones. For example, finding a very detailed simulator which lacks of performance, or by the contrary, finding a very flexible simulator that lacks of very high detail, are very common situations.

Due to the purpose of this work, several critical requirements have been assumed to classify the existing simulators. Those requirements are divided in two groups: functional requirements and architectural design requirements. Moreover, the license of each simulator has been also considered.

Functional requirements involve the support for simulating a corresponding service such as CPU, memory, network and I/O. Following are detailed each one of those services:

- **CPU simulation**. This feature means whether the corresponding simulator is able to simulate the processing system.
- **Memory simulation**. This feature means whether the corresponding simulator is able to simulate the memory system.
- **Network simulation**. This feature means whether the corresponding simulator is able to simulate the network system.
- **I/O simulation**. This feature means whether the corresponding simulator is able to simulate the storage system.

Otherwise, an architectural design requirement involves scalability, flexibility and support for adding new simulators. Following those requirements are explained in detail.

- **Scalability**. This feature means whether the corresponding simulator is able to simulate large-scale systems with enough performance that lets the simulation to be performed in a reasonable amount of time. For example, single processor simulations of highly parallel systems are so slow that researchers must base conclusions on simulations of only fractions of a second of native execution time [SPHC02] [WWFH03b]. Performance determines the amount of workload the model can exercise given the machine resources available for simulation. A common metric is the slow-down, number of simulator host instructions executed per simulated in instruction [FC88]. In general, the more detail that the simulator captures, the greater its slow-down [CK94] [May87]. Slow but accurate simulators have the advantage of capturing subtleties of the target system. However, their slow speed limits the size of the system they can model and the number of simulated instructions they can execute.
- **Flexibility**. This feature means whether the corresponding simulator is able to use several component models with different levels of detail. Also, a flexible simulator must let the user building environments easily by using several component models depending of the user's requirements. The flexibility of integrated simulators is hampered as new devices and new performance models can potentially interact with each other. Complexity and frequent modifications can lead to functional bugs that

are difficult to isolate and fix, as their effect may be detected millions of cycles after they occur. Flexibility indicates how well the model is structured to simplify modification, permitting design variants or even completely different designs to be modeled with ease. Detail defines the level of abstraction used to implement the model's components. A highly detailed model will faithfully simulate all aspects of machine operation, whether or not a particular aspect is important to any metric being measured. The simulator designer must choose a level of simulation detail that is fine enough to capture important performance artifacts, yet fast enough to model large systems and long-running applications in an acceptable timeframe.

- **Support for adding new simulators**. This feature means whether the corresponding simulator is able to use other simulators. By adding new simulators, the corresponding simulation tool will provide a more powerful environment by simulating more services or existing one with more detail.

Table 2.2 shows a comparison between the commented simulators in this section using the previous explained requirements.

Simulator	Functional Requirements				Architectural Design Requirements			License
	CPU	Memory	Net	I/O	Scalable	Flexible	Add simulators	
COTSon	Yes	Yes	Yes	Yes	N/S	N/S	Yes	Open source
Simics	Yes	Yes	Yes	Yes	No	Yes	N/S	Academic
SimOS	Yes	Yes	Yes	Yes	No	Yes	N/S	Research
M5	Yes	Yes	Yes	Yes	No	Yes	N/S	Open source
Bochs	Yes	Yes	Yes	Yes	No	N/S	N/S	LGPL
SimpleScalar	Yes	Yes	Yes	Yes	No	Yes	N/S	Academic
TFSim	Yes	Yes	No	No	No	Yes	Yes	-
UNISIM	Yes	Yes	N/S	N/S	No	Yes	Yes	Open source
SimFlex	Yes	Yes	No	No	No	Yes	N/S	Academic
GEMs	Yes	Yes	No	No	No	Yes	N/S	GPL
Rsim	Yes	Yes	Yes	N/S	No	N/S	N/S	GPL
Talisman	Yes	Yes	Yes	Yes	No	N/S	N/S	-
EPG-Sim	Yes	No	No	No	No	No	No	-
SimUTC	Yes	No	Yes	No	Yes	No	Yes	-
MIDAS	Yes	Yes	Yes	Yes	No	No	No	-
SIM-San v3	Yes	Yes	Yes	Yes	Yes	No	No	Free (binary)

Table 2.2: Comparative of simulators

2.3 Modeling and Simulating applications

Application-architecture combination has widespread applicability in distributed systems research. The results from such an evaluation may be used to: select the best architecture platform for an application domain, select the best algorithm for solving the problem on a given hardware platform, predict the performance of an application on a larger configuration of an existing architecture, predict the performance of large application instances, identify application and architectural bottlenecks in distributed and parallel systems to

suggest application restructuring and architectural enhancements, and evaluate the cost vs. performance trade-offs in important architectural design decisions.

Currently the methods for modeling and simulating applications can be classified in 4 groups: instruction-driven simulation (IDS), execution-driven simulation (EDS), trace-driven simulation (TDS) and distribution-driven simulation (DDS).

2.3.1 Instruction-Driven Simulation

Instruction-driven simulation is an important enabling technology for virtualization [SN05], because the virtual machines must support a program binary compiled for an instruction set that is different from the one implemented by the host processors.

This technique has been developed to simulate the execution of real programs with a high degree of accuracy, where the program to be simulated is stored in the simulator's memory in the same way as it would be in the simulated computer. The simulator program repeatedly fetches instructions from memory, using the op-code to select a routine to execute that will simulate the effects of that op-code. Generally, Instruction-Drive simulation is done interpretively.

An interpreter uses a simple *fetch-decode-execute* loop, a technique that works with all kinds of programs, including self-modifying code. The major drawback is its poor performance, because each instruction has to be decoded over and over again. Figure 2.6 shows the basic schema of this approach.

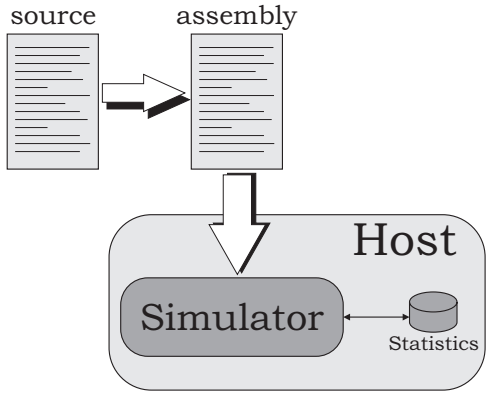


Figure 2.6: General schema of Instruction-Driven Simulation

Currently, there are several approaches for simulating using the instruction-driven technique: interpretation, compilation and binary translation.

Interpretation [LDG+08] is an approach that is potentially much slower than compiled instruction-set simulation [MAF91], which has a start-up cost due to the generation and compilation of the simulator.

Before interpretation of a program, the interpreter will construct an image of the source memory and a data structure called source context block that contains all the registers of the source ISA (Instruction-Set Architecture) to be emulated. When emulation starts, the interpreter fetches the first instruction from the source binary; then extracts the op-code from the instruction; according to the op-code, the interpreter will dispatch a specific

function that emulates the fetched instruction by performing computation and assigning correct values to the source context block and the source memory. The interpreter iterates all the instructions of the source binary, and performs the *fetch-encode-dispatch-execute* procedure for each instruction.

An interpreter spends the majority of its time fetching and decoding the operations, whereas a compiled simulator spends most of its time in performing the computation. The relative cost of interpretation as opposed to computation depends on the complexity of the instruction, addressing modes, etc. Typically, simulators based on interpretation techniques execute from ten to a hundred instructions for each interpreted instruction.

Compiled instruction-driven simulation has some limitations, such as its inability to run programs containing self-modifying code or dynamically loaded libraries. Also, the start-up cost is often seen as a major drawback and has limited the adoption of compiled instruction-set simulation.

There are some works in literature, like [AB02] focused on alleviating this start-up cost. The authors of this work present BSCISS, a generator of compiled instruction-set simulators that works at the assembler level. This approach allows building a system that combines flexibility, accuracy and very fast simulation, along with a small start-up cost. BSCISS automatically generates compiled simulators from a description of the target architecture. At present, this approach allows targeting various statically scheduled RISC and VLIW processors. Within this kind of architectures, the simulators generated by BSCISS are cycle-accurate. That is, the simulator outputs the exact number of cycles needed by the target processor to run the program. Caches can be simulated by interfacing to an external module. Other architectures can be simulated at a functional level, which is only the behavior of the program that will be simulated.

The work [RBMD03] presents a re-targetable simulation framework that supports many variations of architectures with any instruction-set complexity while generating high performance ISA simulators. To achieve maximum re-targetability, a generic instruction model has been developed to be coupled with a decoding technique that flexibly supports variations of instruction formats for widely differing contemporary processors. This model can also be used to exploit all possible instruction formats to generate optimized code for them. This generic model has been used to capture the behavior and binary encoding of the instructions.

The EXPRESSION Architecture Description Language (ADL) [HGG+99] is used to capture the structure of the architecture. In this work the authors use a framework that performs the Instruction-Set Compiled Simulation (IS-CS) technique to generate fast and flexible simulators by automatically generating the instruction templates from the descriptions.

Binary translation [SCK+93] works in a philosophy different from interpretation. The basic idea of binary translation consists on converting a binary program from the target architecture to the host's instruction-set. The translation can be either static [SBR05], when the whole program is processed before execution, or dynamic [UC00] [GFP09] when instructions are translated *on-the-fly*. In binary translation, the generation of target code is optimized by a direct mapping of target ISA registers to source ISA registers. The mapping eliminates the source context block data structure.

Static binary translation is similar to compiled instruction-driven simulation in that

the whole target program is translated once. But instead of directly generating a binary, a compiled simulator generator produces a high-level language program implementing the target program's behavior. This program is then compiled using the host compiler. This makes compiled simulation independent from the host architecture, and allows relying on the host compiler to perform low-level optimizations.

Currently there is a wide range of fields where Instruction-Drive Simulation can be used. One of the most useful contexts where this approach has a remarkable importance is the evaluation of different instruction sets in the early steps on the developing of new computer architectures. Other fields for applying this approach are the validation of compilers, testing, tuning and debugging programs, on a user friendly PC or workstation rather than on actual processors which might not even exist yet.

2.3.2 Execution-Driven Simulation

Execution-driven simulation techniques consists on modifying the application codes by inserting additional instrumentation code, generating input codes which to be executed directly on a simulated environment where all the system services are simulated and the real execution time is measured and transformed into simulated time. Those executions cause the imitation of the behavior of the original applications events reflecting executing on a modeled machine. Some works that uses the execution-driven simulation to model applications are [PY93], [RSJC94], [DJS94], [WWFH03a], [FSS00], and [CMMS88].

This requires that a program used to drive a simulation be extensively modified by a profiler before execution, with the purpose to produce execution times estimated at run-time. Generally, during program execution the profiler parses the program's assembly code to extract the corresponding information, like address references that can be used to simulate architecture and cache organization, timing the underlying time, and memory information that is used to update simulation and the type of memory access. Using this technique, the execution of the program and the simulation of the architecture are interleaved. Figure 2.7 shows the basic schema of this technique.

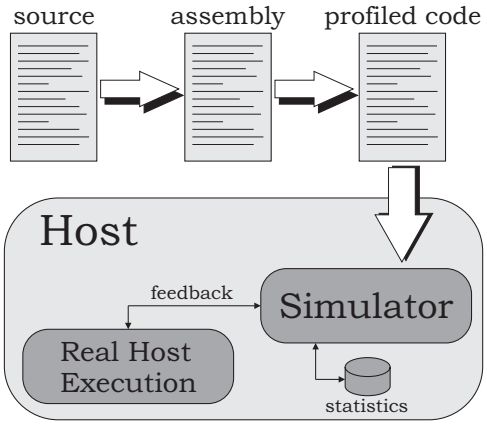


Figure 2.7: General schema of Execution-Driven Simulation

Following, the fundamentals of the execution-driven simulation technique are listed:

- All the system services are simulated and, while they provided real data and per-

form the real service, the execution times are simulated times that depend on the underlying architecture that is being simulated.

- The code of the application is executed on a real CPU and the time expended in this execution is measured and accounted as simulated time.
- The memory used can be also accounted and included in the simulation.

Basically there are two ways to implement execution-driven simulated applications:

First approach consist on obtaining the real application, compiling it using the simulation framework and executing it onto the simulated architecture. This is the preferred option but it can only be used if the simulation framework covers this possibility.

The second approach is more frequent and consists on adapting the real application as much as required in order to make it ready to be compiled on the simulation framework. The required modification can vary from a slight change to a complete rewrite of the application. The most important part of the execution-driven applications is the way system services are simulated and the way CPU and memory are accounted.

2.3.3 Trace-Driven Simulation

In Trace-driven simulations an observer program traces the instructions executed and the data referenced while an observed program is being executed. Collecting detailed traces is extremely costly. Generally those traces need a considerable amount of space to be stored. Also, in most cases generating the trace implies an overhead in the program execution, obtaining times that do not correspond exactly with the real program execution. Thus, it is desirable that the used method for obtaining the program's trace does not interfere with the program's execution and generates a light trace which does not require high storage requirements. Currently exist tracing techniques that make it possible for anyone to trace a program simply and economically, like [CK05] and [Lar93]. Figure 2.8 shows the basic schema for this approach.

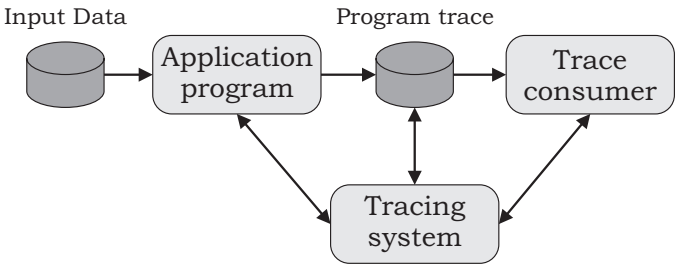


Figure 2.8: General schema of Trace-Driven Simulation

It's important to note the difference between program tracing and program profiling, which must not be confused. Tracing is aimed at generating a listing of the program instructions and data-reference addresses in the proper run-time order. Profiling is a much quicker process which measures or samples the execution frequency of program statements; it ignores data references, recording only an aggregate count of the number of times each statement executes.

The accuracy of such simulations in predicting the system performance is primarily determined by the accuracy of the machine model, and the accuracy of the trace inputs at representing the intended applications. Thus, trace-driven simulation is a widely used technique for evaluating different system design options.

Generally there are two steps that must be performed in trace-driven simulation. First step consists on collecting the events and information of interest to generate the trace. Secondly, a model is simulated using the collected trace as the input.

In most cases, for all but the most trivial programs, collecting all the actions issued by a program is usually impractical because of the cost of storing the trace and the time taken to simulate all the events in the collected trace. Besides since the computing resources needed for simulation depend on the size of the trace, it is not always practical to use the complete trace of an application for simulation. Thus, researchers use techniques for obtaining reduced traces, like sampling [SPHC02].

A challenge that has to be addressed is choosing an appropriate subset with the minimum number of instructions to meet a given error bound. Also there are several sampling techniques for obtaining reduced traces. Some of those techniques are:

- **One large sample** [BYP+91][KE91]: A single sample of a chosen size (from less than 1 million to 10 million instructions) has been used to reduce full size traces (of few billion instructions). This early approach to trace size reduction was later modified to skip the first few (several million) instructions to obtain a more representative sample. Performance projections of several processor models are based on such sampling.

- **Multiple periodic samples** [FP94] [LPI88]: This approach is based on stitching together several contiguous samples. The interval between samples is fixed and usually adjusted to cover the full trace.

- **Multiple random samples** [Lau94]: This is similar to the previous approach except that the inter-sample interval length is random instead of fixed.

A reduced size trace differs from the full trace in the following two aspects: First, there are instructions in the full trace which are simply not present in the short trace. Second, the instructions present in the short trace may be in a different context than in the full trace.

2.3.4 Distribution-Driven Simulation

Distribution-driven simulations use a statistical model of the program to drive the simulation. This model usually takes the form of one or more random processes that model the generation of the data that is transferred between the modules of a system during the execution of a program.

Detailed simulations need long execution times which in most cases are many orders of magnitude slower than real time executions, making it difficult to evaluate numerous design alternatives. Thus, for quick simulation and obtain a preliminary results analytical simulations [WGP09] [KAH+01] becomes very useful. The main advantage of these approaches is that simulations are usually much faster than with an approach that simulates

the internal details of the system's behavior. The main disadvantage of this technique is the potential inaccuracy of the results.

Some simulations take as input only fixed non-random values, typically representing parameters that describe the model and the particular variant of it which is being simulated. This model is called deterministic simulation model. The more interesting aspect of this model is that since there is no randomness in the input, there's no randomness in the output either. Thus, if the simulations are repeated several times, the obtained results will be always the same. The inputs are thus (deterministic) values, and the outputs are the (deterministic) performance measures obtained by transforming the input via the simulation's logic into the output. But many systems involve some kind of uncertain, random input. These models are called stochastic simulation models. The purpose of such a simulation is to learn (infer) something about these unknown output distributions, like maybe their expected values, variances, or probabilities on one side of some fixed tolerances.

Another issue that has an important influence on the simulation model is whether time plays a role in the system. Some simulations don't involve the passage of time, and are called static, like Monte Carlo evaluation of integrals [Ueb97]. The *design-and-analysis* approach is conceptually simple: repeat, or replicate, the model as many times as necessary to get the required precision. Methods from classical statistical analysis can usually be used directly. But most simulations of industrial interest involve the passage of time as an important element; these are dynamic simulations, and the design-and-analysis approach can be a lot harder.

Statistical simulation can be used in combination of other simulation techniques. After an initial detailed simulation, during which program statistics are collected, it is possible to evaluate the system design variations two to three orders of magnitude faster than with conventional detailed simulation, while losing the corresponding percent of accuracy. Consequently, statistical simulation is a useful technique for narrowing the design space during the early phases of design. For instance, many researches combine this technique with the trace-driven simulation. The basic idea is to obtain a statistical result of a very detailed simulation or even from real executions and then generate a synthetic trace using this information. This technique is also called statistical profile simulation [EdBN00].

First, a statistical profile or a set of statistical program characteristics is extracted from a program execution. This statistical profile is then used to generate a synthetic trace which is subsequently fed into a trace-driven simulator, which will estimate the attainable performance for the modeled system. A statistical profile includes many relevant properties of a benchmark execution except for dynamic properties. This approach has two major advantages. First, due to the statistical nature of the synthetic trace generation process, performance characteristics will quickly converge, and hence the number of clock cycles to simulate can be limited. As a result, this methodology can be used to perform a quick design space exploration in an early design stage. Second, by assuming statistical independence of various program characteristics, the statistical profile will be much more compact than a trace, and does not depend on the size of the trace. Figure 2.9 shows the basic schema of this approach.

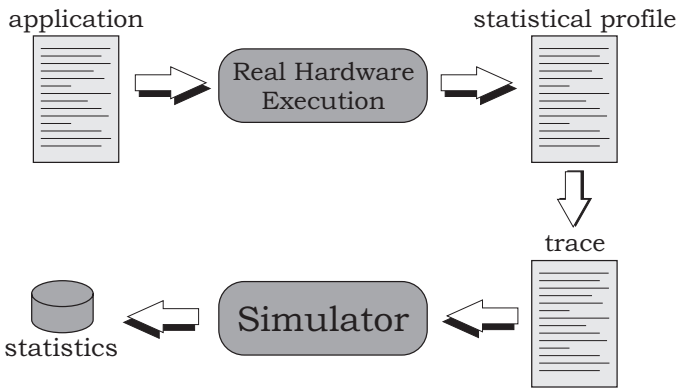


Figure 2.9: General schema of statistical profile simulation

2.4 Modeling and Simulating High Performance Applications

Simulating HPC applications is very useful to analyze, debug, and predict the performance of parallel programs for a variety of parallel architectures. Nowadays there are a high number of works for modeling and simulating HPC computing applications for both architecture models: message passing and shared memory.

Performance modeling is a key approach that can provide information on the expected performance of a workload given a certain architecture configuration. It is useful throughout a system life-cycle: starting at design when no system is available for measurement, in procurement for the comparison of systems, through to implementation and installation, and to examine the effects of updating a system over time. A model adds insight into the performance of current systems, reveal bottlenecks and show where tuning efforts would be most effective. They also allow the performance on future systems to be explored.

Dimemas [LGP+96] [GLB00] is a trace driven performance prediction tool for message passing programs, which enable users to develop and tune parallel applications by giving accurate prediction of their performance on the target machine architecture. Supported architectures include networks of workstations, shared memory processors (SMP) and clusters. Dimemas rebuilds the behavior of a parallel program based on an input Dimemas trace file and information of the supported target machine architecture. Then, it generates a visualization trace file that is viewed using a visualization tool such as Paraver [LGP+96]. The user can then analyze the performance and modify the message passing programs. Dimemas models the target machine architecture as a network of nodes where each node is an SMP connected to the network with a set of links and buses.

MPISim [PB98] is a library incorporated in the message passing interface (MPI) standard to enable prediction of the performance of MPI programs based on architectural characteristics, such as number of processors and message communication delays. MPISim assumes that the program has no I/O commands and simulates all collective communication functions in terms of point-to-point communication. All point-to-point communications are implemented using a set of four core non-blocking MPI functions. In order to enable the MPISim simulation to be run on a single processor machine, there is a need to modify an existing MPI program to support multithreaded execution. A pre-processor is provided with MPISim to automatically privatize permanent variables, changes each MPI call to

MPISim call and implements various transformations needed to link the program with the MPISim library. During simulation, each process in the MPI program is defined as a logical process (LP) in MPISim. Sequential code blocks are simulated through direct execution, while each call to a MPI communication function is translated to a corresponding MPISim function. The translation is done internally by MPISim to replace MPI functions to a set of four core non-blocking MPI functions.

The work [BDP01] extends MPI-SIM with MPI I/O functions by using a parallel File System simulator inspired on Vesta [CF96]. This simulator is used to predict the performance of existing MPI programs as a function of several architectural characteristics, including number of processors and message communication latencies. However, this simulator only provides a specific architectural model with little flexibility.

A simulator for MPI applications is described in [Rie06]. This work is a prototype of a simulator which describes an approach that is a hybrid between running a parallel application in stand-alone mode and simulating the network it uses for MPI data exchanges. This simulator requires for the simulated application to be launched on similar or the same hardware to be simulated; i.e. the same CPU speed and type, same memory subsystem, etc. The early prototype simulator described in this work can only simulate the network, not yet the nodes. Moreover, the simulator does not include MPI-IO functions.

The authors of [BDDP99] describe the use of COMPASS, a portable, execution driven, asynchronous parallel discrete event simulator that can be used to predict the performance of large-scale parallel programs, including computation and I/O intensive applications, targeted for execution on shared-nothing and shared memory architectures, as well as SMP clusters. In particular, simulation modules have been developed to predict the performance of applications as a function of communication latency, number of available processors on the machine of interest, different caching strategies for parallel I/O, parallel file system characteristics, and alternative implementations of collective communication and I/O commands. The simulator is being used for detailed simulations within the POEMS project [ABB+00].

[AS00] presents a common parallel program representation, designed to support such a comprehensive approach, with four design goals: (1) the representation must support a wide range of modeling techniques; (2) it must be automatically computable using parallelizing compiler technology, in order to minimize the need for user intervention; (3) it must be efficient and scalable enough to model teraflop-scale applications; and (4) it should be flexible enough to capture the performance impact of changes to the application, including changes to the parallelization strategy, communication, and scheduling. Analytical models in predicting performance of the Sweep3D application on very large machine configurations are shown in [SSV99].

The framework [SCW+02] combines tools for gathering machine profiles and application signatures and provides automated convolutions. Convolution methods are techniques for mapping signatures to profiles in reasonable time complexity for predicting and understanding performance. The convolution method presented in this work consists on mapping an application's signature (application A) onto a machine profile (machine B) to arrive at a performance prediction (performance of application A on machine B). Machine profiles are tables of performance data gathered for existing machines via low-level benchmarks that measure simple performance attributes of machines. For machines that have yet to

be built, machine profiles represent the engineer's expectations of the rates at which the machine will perform operations. An application signature is a summary of the operations required by an application to accomplish its computation.

This work [MANR09] addresses the utilization of traces taken from MPI applications to do simulation-based performance studies of parallel computing systems. The authors of this work use several mechanisms to capture traces, pointing out important limitations of some of them. One of these limitations is the invisibility of message interchanges in collective operations, which is circumvented modifying a trace-capturing library. During a simulation, trace records must be simulated in causal order, to fully comply with application semantics. The techniques introduced in this work have been implemented in the INSEE [PMA05] simulation environment, which is used in two example studies to show its usefulness: an evaluation of alternatives for interconnection network design, and a performance prediction study in which traces from one machine are used to estimate the execution times of applications running in a different machine.

Programs written using two programming models, such as MPI and openMP, require an analysis to determine both performance efficiency and the most suitable numbers of processes and threads for their execution on a given platform. The work [AC06], in order to study those problems, proposes the construction of a model that is based upon a small number of parameters, but is able to capture the complexity of the runtime system. This work aims to model hybrid MPI and openMP programs analytically to detect communication and parallelization inefficiency, as well as inefficiencies caused by the strategy used to combine the two programming models, and lastly to use the described model and additional insights to optimize the performance of the mixed mode model.

Analytical techniques for predicting detailed performance characteristics of a single shared memory parallel program for a particular input are studied by the POEMS project in [AV04]. The authors of this work have developed an accurate performance model for parallel applications executing on dedicated, shared memory systems. The model has two levels: a lower-level queuing model to characterize the impact of contention and caching effects, and a higher-level task graph model of the application.

[KRR04] describes a compiler tool to automate performance prediction for execution times of parallel programs by runtime formulas in closed form. For an arbitrary parallel MPI source program the tool generates a corresponding runtime function modeling the CPU execution time and the message passing overhead. The environment is proposed to support the development process and the performance engineering activities that accompany the whole software life cycle.

The Performance Analysis and Characterization Environment (PACE) [NKP+00] developed by the High Performance Systems Group at the University of Warwick is a performance prediction system that provides quantitative data concerning the performance of (typically scientific) applications running on high performance parallel and distributed computing systems. The system works by characterizing the application and the underlying hardware on which the application is to be run, and combining the resulting models to derive predictive execution data. PACE provides the capability for the rapid calculation of performance without sacrificing performance accuracy. PACE also offers a mechanism for evaluating performance scenarios, for example the scaling effect of increasing the number of processors, and the impact of modifying the mapping strategies (of process to processor)

and underlying computational algorithms.

BigSim [ZKK04] is a Parallel simulator for predicting performance of machines with a very large number of processors. The simulator provides the ability to make performance predictions for machines such as BlueGene/L, based on actual execution of real applications. Based on the CHARM ++ [KK96] parallel programming system, this emulator has successfully emulated several million threads (one for each target machine processor) on clusters with only hundreds of processors. However, the emulator is useful only for studying programming models and application development issues that arise in the context of such large machines. Specifically, the emulator does not provide useful performance information.

Essentially, this approach involves letting the emulated execution of the program proceed as usual, while concurrently running a parallel algorithm that corrects time-stamps of individual messages.

Based on the low level programming API provided by the emulator, several parallel programming languages are implemented on BigSim. They are MPI [GHLL+98], CHARM ++ [KK96] and Adaptive MPI [HLK03].

The authors of this work [KWH02] show how by the examination of the key characteristics of an application, analytical performance models can be formed. These models are parameterized in terms of computational and communication performances of an individual system and can be used to explore achievable performance of an application prior to system availability. One of the models is utilized to validate the performance of a Compaq Alphaserver ES45 supercomputing system being built at Los Alamos, and expected to grow to 30 Tera-flops peak performance. The approach described is application centric. This involves the understanding of the processing flow in the application, the key data structures, and how they use and are mapped to the available resources. From this a performance model is constructed that encapsulates its key performance characteristics.

2.5 Performance analysis tools for parallel systems

Apart from simulators there are other tools for analyzing and studying the performance of parallel applications, called performance analysis tools. With the ever increasing complexity in the current High Performance Systems, detecting bottlenecks or poor resource management of parallel applications executed on those systems require an enormous effort. The main objective of those tools is to help to detect bottlenecks and thus ease the performance analysis of an application, which is very useful for fully exploiting the potential of high-performance computers.

The main difference between those tools and simulators is that those tools need the application be executed on the real system, and simulators let the application can be executed in a simulated environment which does not require to have the real system. Currently, several performance analysis tools can be found in literature.

HPCToolkit [ABF+10] is an integrated suite of tools that supports measurement, analysis, attribution, and presentation of application performance for both sequential and parallel programs. HPCToolkit can pinpoint and quantify scalability bottlenecks in fully optimized parallel programs with a measurement overhead of only a few percent. Recently, new capabilities were added to HPCToolkit for collecting call path profiles for fully optimized codes without any compiler support, pinpointing and quantifying bottlenecks in

multithreaded programs, exploring performance information and source code using a new user interface, and displaying hierarchical space-time diagrams based on traces of asynchronous call path samples.

Online tools, such as Paradyn [RM06] and Periscope [GO10], evaluate performance data while the application is still running. Both tools search for previously specified performance problems or properties. A search strategy directs performance measurements, which are successively refined based on the current findings. To ensure scalable communication between tool back-end and front-end, their architectures employ hierarchical networks that facilitate efficient reduction and broadcast operations.

Paradyn was the first tool that automated performance analysis. Its Performance Consultant guides instrumentation and searches for bottlenecks based on summary information during the program's execution. Periscope is a highly scalable tool for the automatic distributed online search for the performance properties of large-scale applications on high-end computers. It allows for both detection of the performance bottlenecks limiting the scalability on parallel systems as well as pinpointing the issues concerning the single-node performance of an application.

Scalasca [GWW+10] is a performance toolset that has been specifically designed to analyze parallel application execution behavior on large-scale systems with many thousands of processors. It offers an incremental performance-analysis procedure that integrates runtime summaries with in-depth studies of concurrent behavior via event tracing, adopting a strategy of successively refined measurement configurations. Distinctive features are its ability to identify wait states in applications with very large numbers of processes and to combine these with efficiently summarized local measurements.

Vampir[RKH+08] is a widely used visualization and analysis tool for parallel applications. Trace files are generated during execution and visualized with a powerful GUI. Yet, huge trace files are generated and have to be analyzed manually afterwards.

2.6 Summary

In this chapter a set of well-known tools and techniques for predicting and analyzing the performance of computer systems has been described. Depending of the requirements and the part of the system to be analyzed, a set of techniques and tools will be chosen for accomplishing this purpose.

In this thesis we propose an approach for simulating both large distributed environments and applications, balancing the trade-off between the speed of simulations and the accuracy obtained. Thus, we need to found a technique or tool that be scalable, fast and accurate enough that satisfies those requirements.

Due to complete distributed systems have to be simulated and modeled, an initial approach is to use existing simulators for modeling and simulating each corresponding part of the distributed system. For instance, using DiskSim [BSSG08] for simulating the disks drives, NS-2 [NS2] for simulating the network system, [CK94] for simulating the CPU, [WGT+05] for simulating the memory, and so on until all parts of the distributed system be simulated. The main advantage of using individual simulators is the flexibility, because simulation can be focused in a set of systems, depending on the requirements of the user. The main problem of this approach is the interoperability. Due to each simulator is written

for a specific platform, which includes operating system, programming language, compiler, etc., integrating a set of simulators for building a unified simulator for simulating complete distributed systems is unfeasible and impractical.

Another approach is using complete computer architecture simulators. Those simulators can simulate and model complete distributed systems. The simulators that provide the best accuracy for this purpose are called full-system simulators. This kind of simulator provides a very high level of accuracy because the complete system is simulated with full level of detail. Even some existing full-system simulators can load unmodified commercial operating systems. The main problem of those simulators is that they need huge amounts of time to execute simulations. In some cases, the slowdown factor of those simulations can vary from five or six orders of magnitude slower than real system, which is not a feasible solution for the purpose of this thesis.

Also, currently there are other non-full-system simulators that can simulate complete computer architectures. Some examples are RSim [HPRA02], Talisman [Bed95], and MIDAS [TSG08]. Those simulators provide better performance than full-system simulators at the cost of sacrificing accuracy. This loss of accuracy is because those simulators do not model the system with full level of detail. As opposed to use individual simulators, the main problem of this kind of simulators is that they are too focused in a specific field, which lack of flexibility for simulating different computer architectures and configurations. Thus, we need a more scalable and flexible tool for accomplishing the purpose of this thesis.

A further approach is to use performance analysis tools. Those tools are used for detecting performance bottlenecks and thus ease the performance analysis of applications, which is very difficult task for parallel applications executed in large environments of thousands of CPU cores. Thus, a very detailed performance analysis can be performed for each system such as I/O, Memory, CPU, and Network when parallel applications are executed in distributed architectures. The problem of using those tools is that the real system which the application is executed is needed. The enormous cost of those systems and the complexity for deploying them by using different configurations are serious limitations that hamper analyzing applications using those systems.

Then, after analyzing the currently existing tools and techniques for simulating distributed systems and applications, the conclusions is that there is no one that satisfies the requirements for accomplish the main purpose of this thesis.

Chapter 3

The SIMCAN simulation platform

This chapter shows a proposal of a fast, flexible, scalable and expandable simulation platform for modeling and simulating distributed systems and applications.

The new contributions presented in this chapter are the features provided by this simulation platform for building simulation models, by defining and configuring each one of the four basic systems independently, which consists of processing system, memory system, storage system and network system.

The main advantage of those contributions is the flexibility and scalability obtained for modeling and simulating large and complex environments. Thus, those environments can contain thousands of nodes, modeling each subsystem with the required level of detail.

Moreover, several methods for increasing the functionality of this simulation platform are described.

3.1 Introduction

The ever-increasing complexity of computing systems has made simulators a very important choice for designing and analyzing large and complex architectures. Due to the high number of topics in the field of computer architecture, developing a universal simulator is impractical and unfeasible. Naturally, researchers want to simulate an entire system with total accuracy, but there are obvious difficulties. Some of those difficulties include high cost, time to completion, specification inaccuracies, and implementation errors.

Each researcher has its own objectives and needs, the same way each simulator is developed for a specific purpose. Many existing simulators are monolithic or designed for a single architecture. As a result, it is difficult to extract a micro-architecture component from the simulator for reuse, sharing or comparison. It is true that researchers try to find the simulator that fits the most with its research, but in many cases researchers can't find that simulator and they have to modify an existing one, or coding a new simulator.

A possible approach for building a simulation platform targeted to model and to simulate distributed systems consists on using single-component simulators. Thus, using a set of those simulators for covering all systems will let model entire distributed systems. The main drawback of this approach is that all those simulators have to work together for simulating the complete system, which implies that those simulators must be integrated

in a single platform. This drawback makes unfeasible this approach due to interoperability and compatibility reasons.

Otherwise, another approach is to use simulators targeted to model systems completely, like full-system simulators. Those simulators let model and simulate complete systems with full detail providing a high level of accuracy. The main issue of those simulators is performance. In many cases, the slowdown factor of models built using this kind of simulators is 5 or 6 orders of magnitude slower than the real system.

In order to accomplish the purpose of this thesis, we propose a fast, flexible, scalable and expandable strategy for modeling and simulating distributed systems, which consists on integrating the model of the four basic systems into a single simulation platform. Those systems consist of storage system, memory system, processing system and network system. Figure 3.1 shows the basic layered schema of those systems.

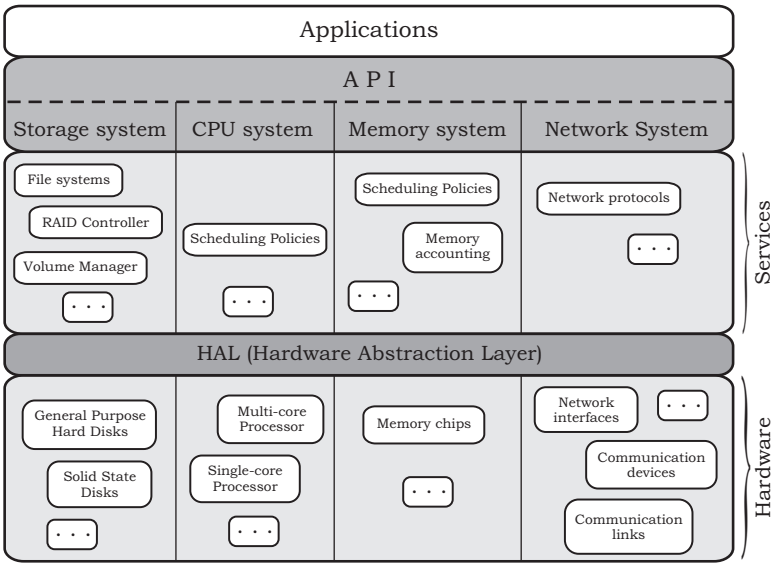


Figure 3.1: Layered schema of the proposed simulation platform

The main objective of this chapter is to propose a strategy for simulating complex and large environments that represent, both actual and non-existent architectures by modeling individually each one of the four basic systems. Then, depending on the user's requirements, the model can be focused on a set of the basic systems, or by the contrary on the complete system. Therefore, a complete distributed system can be modeled by integrating those basic systems in the model, each one with the corresponding level of detail, which provides a high level of flexibility.

The philosophy of this strategy is not to obtain a perfect accuracy, but allowing a minimal margin of error in benefit of executing simulations much faster. Absolute accuracy is not always strictly necessary and in many cases it is not even desired, due to its high engineering cost. In many situations, substituting absolute with relative accuracy between different timing simulations is enough for users to discover trends for the proposed techniques. Normally, speed and accuracy are inversely related. If one of these characteristics increases, the other one decreases. For this reason, the proposed strategy try to find an adjustment between those features, performing quickly simulations with a minimal

percentage of error.

Those proposed features are desirable for any simulation strategy, but their meaning can be blurry depending on the context in which they are used. Following, in the context of this thesis, the meaning of those features are described.

Scalability means whether the corresponding strategy is able to simulate with enough performance large-scale systems, by increasing the number of machines, which make up the corresponding architecture to be simulated. Likewise, performance determines the speed which a simulator executes a corresponding simulation. In general, the larger the size of the architecture to be simulated, the greater the time needed to execute the simulation.

A flexible strategy must let users build environments easily, using several component models with different levels of detail. Flexibility also indicates how well the model is structured to simplify modification, allowing design variants or even completely different designs to be modeled smoothly.

Detail defines the level of abstraction used to implement the model components. Thus, researchers can choose a level of detail for modeling the required architecture that is fine enough to capture important performance artifacts, yet fast enough to model large systems and long-running applications in an acceptable timeframe. In general, the more detail the simulator captures, the greater its slow-down.

Finally, the term expandable refers to the ability of increasing the functionality of a simulation platform. In most cases, this ability consists on adding new models to the repository of the simulation platform.

Due to nowadays there is no simulation platform that provides those features required to perform the strategies proposed in this thesis, a new simulation platform called SIMCAN, that contains the strategies proposed for modeling and simulating large distributed environments, has been designed and developed.

3.2 System architecture

Nowadays the main trends go towards Chip Multi-Processors (CMPs), architectural customization, and heterogeneity, which imply that the focus of simulation will shift from in-core behavior to system behavior. Thus, keeping in mind those trends, the proposed simulation platform has been designed to perform those kinds of simulations.

SIMCAN has been designed targeting to provide flexibility, accuracy, performance and scalability, which makes it a powerful simulation platform for designing, testing and analyzing both actual and non-existent architectures. The range of systems to simulate comes from a single computing node to a complete high performance distributed system. The best asset of this simulation platform is that SIMCAN is able to model and simulate large environments (thousands of nodes) with a customizable level of detail [NnFG+10]. Moreover, this simulation platform has been applied to data systems simulation in the EPCC art Edinburgh University [FHN+09].

Simulated architectures are modeled using a set of existent components provided by SIMCAN; they represent the behavior of real components that belongs to real architectures like disks, networks, memories, file systems, etc. Those components are hierarchically organized within the repository of SIMCAN, which made up the core simulation engine.

Currently, SIMCAN provides a wide range of components to model a complete distributed system with different levels of detail and scalability.

Besides designing simulated environments using components provided by SIMCAN, new components can be added to its repository. Moreover, SIMCAN allows an easy substitution of components for a particular component, such as a Disk, within a particular behavior. Those interchangeable components can differ in level of detail (to make performance versus accuracy trade-offs), in the functional behavior of the component, or both. Furthermore, new and existing simulators can be added to the repository of SIMCAN, like diskSim [BSSG08]. This process is explained in detail in sections 3.5. This entails a powerful feature for the simulation platform because once new components are added to the repository of SIMCAN, its functionality increases as well.

The way SIMCAN is executed for performing a corresponding simulation will depend both of the user requirements and the resources available. Therefore, SIMCAN can be executed in a single computer using sequential simulation or, by the contrary, it can be executed in parallel using both shared memory computers and distributed memory systems. The speed of the simulation will depend highly of the computing resources used for executing the simulation. The more CPU and memory resources available, the better performance will be obtained for executing the simulation. However, the way the simulation is parallelized in the different CPUs depends of the configuration set by the user. A method for automatically creating parallel simulation models is described in detail in section 4.2.

With the purpose of ease the task for building and configuring large distributed environments, the SIMCAN platform provides a flexible classification of node aggregation blocks that mimics aggregations used in real systems. Following is described a collection of node aggregation blocks provided by SIMCAN, which let users create the most commonly used architectures:

- **Compute Node**: This module simulates the behavior of a node. It contains the necessary modules to simulate the required systems (see figure 3.1) that are included in a real node. The components of each node can be fully customized and configured to act as a computing node, as a storage node or as a mix of computing and storage node.

- **Node Board**: This module is an aggregation of several nodes that are locally arranged in terms of communication procedures. For this reason node boards include a local switch that connects all its nodes, acting as the only communication port outside the board. The number of nodes and the characteristics of the switch are fully customizable by the user.

- **Rack**: This module is an aggregation of several node boards used for configuration and managing purposes. Each node board in the rack has its own communication channel. The number of node boards is fully customizable by the user.

Figure 3.2 shows the basic aggregations described before. The rest of the architecture (mainly the storage nodes and the rest of the communication switches) can also be grouped in different aggregations. However, the module aggregation is not limited to the modules presented in this section. New module aggregations can be defined and added to

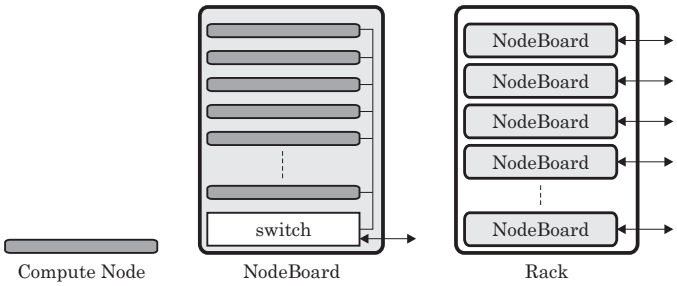


Figure 3.2: SIMCAN packaging

the repository of SIMCAN increasing the range of possible configuration for developing new architectures and environments.

In a computer system, the node is the most relevant component. Similarly, in SIMCAN a node is basically a building block for creating distributed environments. Basically, a distributed environment consists of nodes, communication devices like switches or routers, and communication networks. Moreover, the architecture of SIMCAN is flexible and it does not restrict only the use of existing nodes for building environments. In a SIMCAN node, both the four basic systems and all applications are connected to the API module (see figure 3.3).

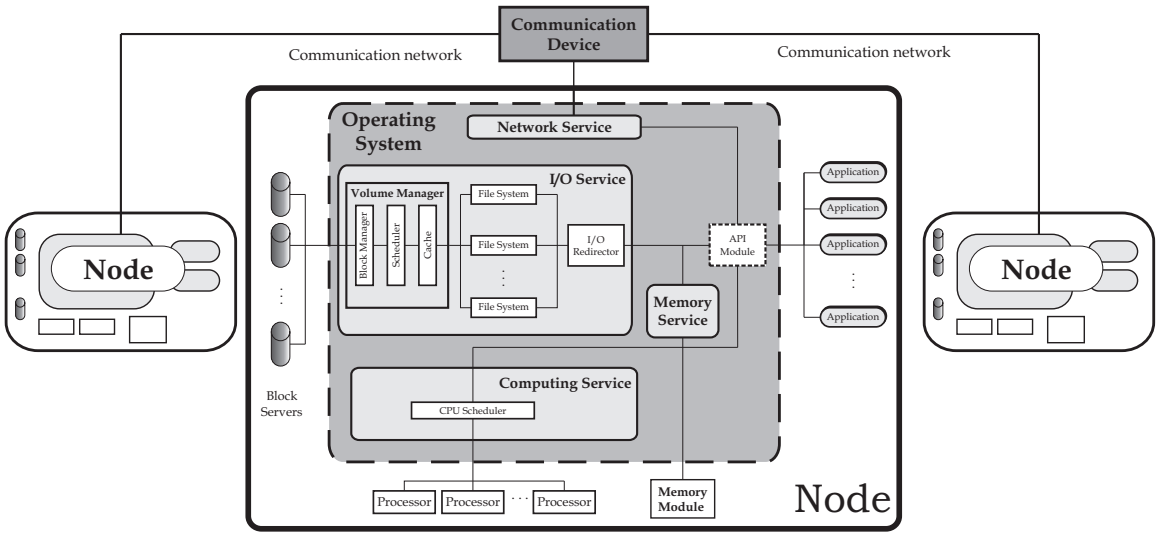


Figure 3.3: Global architecture of SIMCAN

The SIMCAN simulation platform has been built on the top of INET and OMNeT++ frameworks. Moreover, other simulators can be added to the framework architecture to increase its functionality. Figure 3.4 shows the framework layers and the interaction between them.

3.2.1 Features

The most remarkable features of the SIMCAN simulation platform include the following:

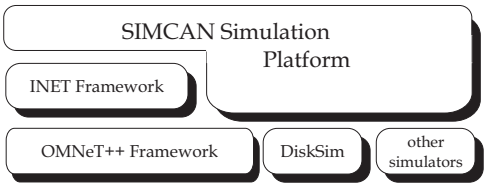


Figure 3.4: Framework layers

- Both existing and non-existing architectures can be modeled and simulated.
- It provides a POSIX-based API for simulating new applications.
- It balances the trade-offs of scalability, performance and accuracy for simulating quickly large environments with a reasonable level of detail.
- It allows automatic partitioning of large models for performing parallel simulation.
- A node can be fully customized by defining each system independently, which include:

  Computing system.

  Memory system.

  Storage system.

  Networking system.

- Computing system can simulate quickly both uni-core/multi-core systems using several scheduling policies.
- Memory system is very useful for counting the amount of memory required by applications for different purposes, like: code, local and global variables, dynamic variables, disk cache.
- It contains a very detailed storage system, which include models for:

  Modeling general purpose file systems like Ext2 and Reiser FS.

  Modeling parallel file systems, like PVFS.

  Modeling general purpose hard disk drives.

  Modeling RAID systems.

  Modeling remote file systems, like NFS.

- Network system can be modeled for simulating a wide range of distributed environments using several levels of detail:

  Low level of detail, where each hop is accounted using its own latency, bandwidth and processing time.

  High level of detail, using the INET framework that also let us to model protocol stacks and data collisions.

- The same architecture can be modeled with different levels of detail.

- It allows several methods for simulating applications like:

  Using traces of real applications.

  Using a state graph [NnFG+10].

  Programming new applications directly in the SIMCAN platform.

- It provides an adapted MPI library for modeling and simulating MPI applications.
- It includes a wide range of components for building and customizing a wide variety of architectures.
- New components can be added to the repository of SIMCAN.
- Both existing and new simulators can be integrated with the simulation platform in order to simulate concrete parts of the modeled architecture.

3.3 Modeling the basic systems using the SIMCAN platform

In SIMCAN, the module in charge of simulating the operating system contains the models of the software components for managing the corresponding basic systems. Those basic systems are: computing system, memory system, storage system, and network system. Outside the operating system module are the components that model the hardware parts, like the processor, the network, the memory and storage devices (see figure 3.3). Each basic system managed by the operating system module is treated independently. Therefore, the selection of the corresponding service required by each application request is performed by the API module.

3.3.1 Strategies for modeling the computing system

The computing system has been modeled in SIMCAN using 2 different components: the processor and the CPU scheduler. In one hand, the hardware part of the computing system corresponds with the processor. In the other hand, the software part corresponds with the CPU scheduler.

The strategy used for modeling this system is based on calculating the amount of time needed for executing the concrete instructions invoked by applications. Thus, those instructions might not be executed in a CPU. Therefore, CPUs are parameterized with a concrete CPU processing power measured in MIPS (Million Instructions Per Second), which is a method of measuring the raw speed of a computer's processors. Similarly, the amount of computing invoked by applications is measured in MIs (Million Instructions).

The fastest method for calculating the amount of instructions executed by a given application consists on performing next steps:

1. Executing the application to be modeled in a concrete CPU.
2. Calculating the amount of time needed for executing that application.
3. Multiplying the amount of time needed for executing the application by the number of MIPS of the CPU used in this process.

This method can be performed both off-line and on-line. First option consists on executing this method separately of the simulation platform. Then, once the corresponding parameters of the application to be modeled have been calculated, the application can be configured properly in a modeled environment. By the contrary, the second option consists on performing this method in the same hardware and at the same time that the simulation is being executed. Then, a set of instructions are executed in a real CPU, whereof an estimation of the number of instructions executed can be calculated easily on-the-fly, before send the corresponding request to the computing system in the simulation platform.

The processor is the component in charge of calculating the amount of time spent in executing instructions invoked by applications. Each processor consists of a finite set of CPU cores, where the speed of all those cores is the same that the speed of its processor. Thus, several applications can be executed in parallel using the same processor, even when the number of applications executed at any given instant is greater than the number of CPU cores. Therefore, processor P can be defined as a finite set of CPU cores such that:

$$P = \{c_1, c_2, c_3, ..., c_n\}$$

Each application contains a set of operations to be executed, which can be categorized in computing operations and blocking operations. Computing operations consist of a set of instructions to be executed in a CPU core. Otherwise, blocking operations are those operations that are not executed by a processor, but another system such storage system or network system. Those operations, sorted following the execution order, are grouped in blocks such that all operations located in the same block are of the same kind.

Figure 3.5 shows the basic schema of how the computing system is modeled in the SIMCAN simulation platform. In this figure, applications contain both computing blocks (white-light grey partitioned blocks) and blocking operation blocks (dark grey). When an application has to execute a computing block, it sends the request to the computing system. When the request is executed, then the computing system returns the control to the application to execute the next block. The computing system is in charge of managing those blocks and executing them in the corresponding CPU cores. By the contrary, blocking operation blocks will be sent to the corresponding system to be processes accordingly.

In a computing system, a tick can be defined as the maximum amount of time that each computing block can be processed by a CPU core continuously, without being interrupted. Similarly, IPT (Instructions per Tick) can be defined as the number of instructions that can be executed in one tick. Therefore, a computing block is partitioned in portions, where each one contains at maximum $IPTs$ instructions. Then, the number of portions of a given computing block Cb executed in a CPU core $CPUcore$ can be calculated using equation 3.1.

$$\left\lceil numPortions(Cb) = \frac{Cb_{MIs}}{CPUcore_{MIPS} \cdot CPUcore_{tick}} \right\rceil \tag{3.1}$$

where $CPUcore_{MIPS}$ is the speed of the core where Cb is executed (measured in MIPS), $CPUcore_{tick}$ is the amount of time spent by one tick in $CPUcore$, and Cb_{MIs} is the number of instructions contained in Cb measured in MIs. Then, computing block is defined as a finite set of portions such that:

$$Cb = \{p_1, p_2, p_3, ..., p_k\} | \forall k (0 < k \leq numPortions(Cb))$$

where each one of those portions can be in two different states: processed or pending, such that $state(p_i) = processed|pending$. Then, the state of a given p_i will be *processed* when p_i had been completely executed in a CPU core. Otherwise, the state of p_i will be *pending* when p_i had not been executed yet. Figure 3.5 shows several computing blocks divided in portions. White portions represent parts of computing block that have not been executed yet. Light grey portions represent parts of computing block that have been already executed. Black portions represent parts of computing block that are currently being executed in a CPU core.

Then, the total time needed for executing the computing block Cb in the CPU core $CPUcore$ is given by the equation 3.2.

$$time_{exec}(Cb, CPUcore) = numPortions(Cb) \cdot CPUcore_{tick} \tag{3.2}$$

The remaining time of a given computing block Cb to be completely executed in $CPUcore$ is calculated using equation 3.3.

$$time_{remaining}(Cb, CPUcore) = \sum_{i=0}^{|Cb|-1} CPUcore_{tick} | \forall p_i \in Cb, (state(p_i) = pending) \tag{3.3}$$

Otherwise, the CPU scheduler is the component in charge of managing all computing blocks to be executed in the corresponding CPU cores efficiently. Thus, this sched-

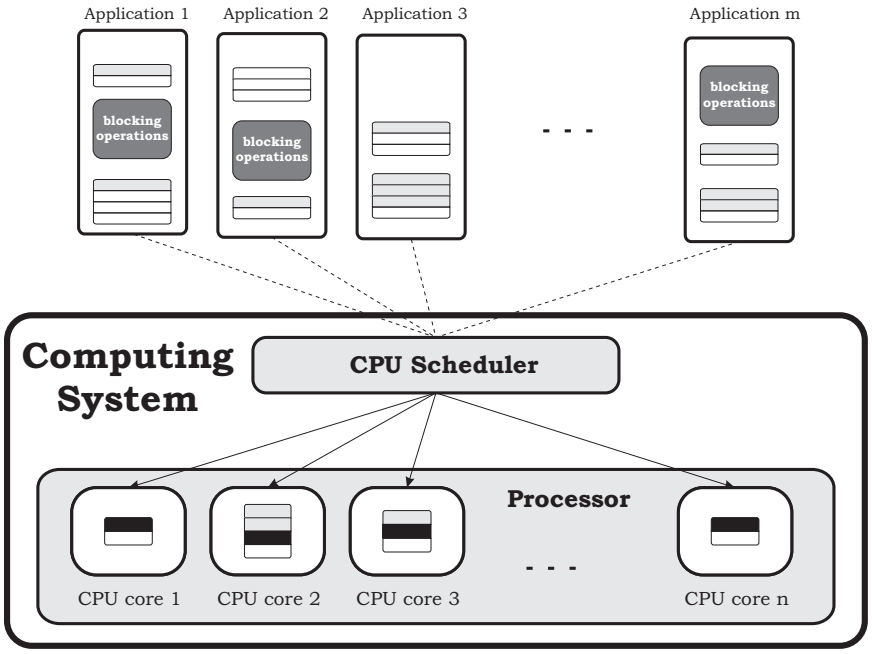


Figure 3.5: Basic schema of the computing system in SIMCAN

uler achieves the effect of an apparent simultaneous execution of multiple applications by switching from one computing block to another.

In SIMCAN, the CPU scheduler has been modeled using the time sharing technique, where several applications simulate their executions by multiplexing the CPU time because the computing blocks belonging to the simulated applications are divided in portions (see figure 3.5). Therefore, in single-core processors, only one application can be executed at any given instant.

The policy for managing computing blocks in the CPU scheduler is fully customizable. In this chapter, three different strategies for managing blocks in the simulated CPU scheduler are proposed. However, new strategies can be added to the simulation platform.

3.3.1.1 CPU Scheduler in SIMCAN using a Round-Robin strategy

This strategy is based on using one run queue shared for all CPU cores, such that the computing blocks sent by applications are managed by this queue. Thus, CPU scheduler uses a round-robin algorithm for sending the first computing block in the queue to a idle CPU core. Then, each core executes slices of the computing blocks in the order that they are found on the queue, with equal amounts of time allowed. If a computing block uses up the time quantum it is allowed, it is placed at the end of the queue, and the first computing block in the run queue is selected to be executed. The quantum is the maximum number of ticks that each computing block can be processed by a CPU core, before the CPU scheduler calculates the next computing block to be processed in the same CPU core.

The configuration of the quantum duration is critical for system performance. In one hand, for short quantum sizes the system can suffer excessively high overheads caused by continuous switching of computing blocks. In the other hand, for large quantum the applications no longer appear to be executed concurrently. Although the quantum size is customizable in SIMCAN, it takes a default value of 0.1 seconds.

Figure 3.6 shows the basic schema of a complete CPU system modeled using this strategy. This figure shows a CPU scheduler with one run queue that contains computing blocks sent from applications. This queue is associated to all CPU cores located in the processor. Each computing block is divided in slices, where each slice corresponds with the quantum. White slices represent parts of computing block that have not been executed yet. Grey slices represent parts of computing block that have been already executed. Black slices represent parts of computing block that are currently being executed in a CPU core.

In this example (see figure 3.6) a new computing block arrives to the CPU scheduler. Then, this computing block is placed at the end of the queue. Initially, all its slices are white because this computing block has not been executed yet.

Algorithm 1 shows the algorithm for performing this strategy in the CPU scheduler simulated in SIMCAN.

3.3.1.2 CPU Scheduler in SIMCAN using a FIFO strategy

Another approach for simulating the CPU scheduler consists on using the FIFO policy (First In, First Out). This policy can be considered the same that the Round-Robin but using an infinite quantum. Therefore, each computing block that reach a CPU core is

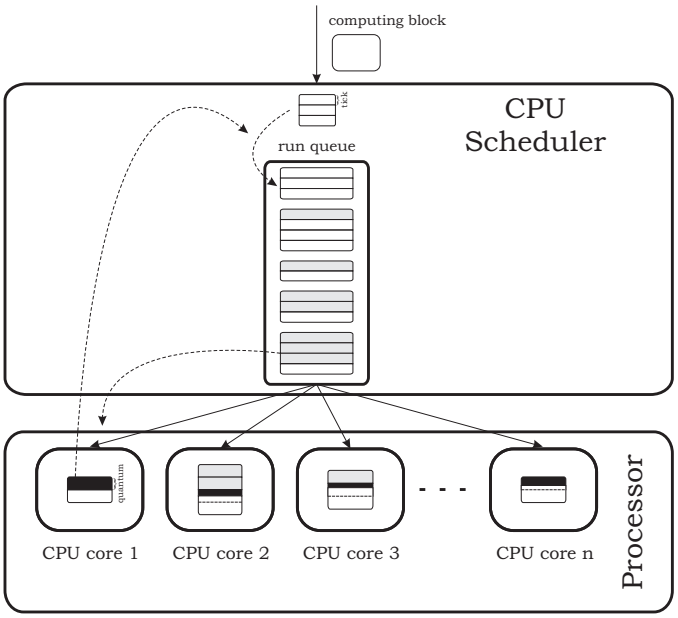


Figure 3.6: Example of a CPU scheduler using a Round-Robin strategy in SIMCAN

completely executed before abandon the CPU core.

Figure 3.7 shows an example of a CPU scheduler using a FIFO policy in SIMCAN. In this figure, white parts of computing blocks represent instructions that have not been executed yet. Otherwise, black parts represent instructions that have been already executed in a CPU core. Computing blocks that finish their executions (see CPU core 2 in figure 3.7) are sent back to the application that sent them to computing system.

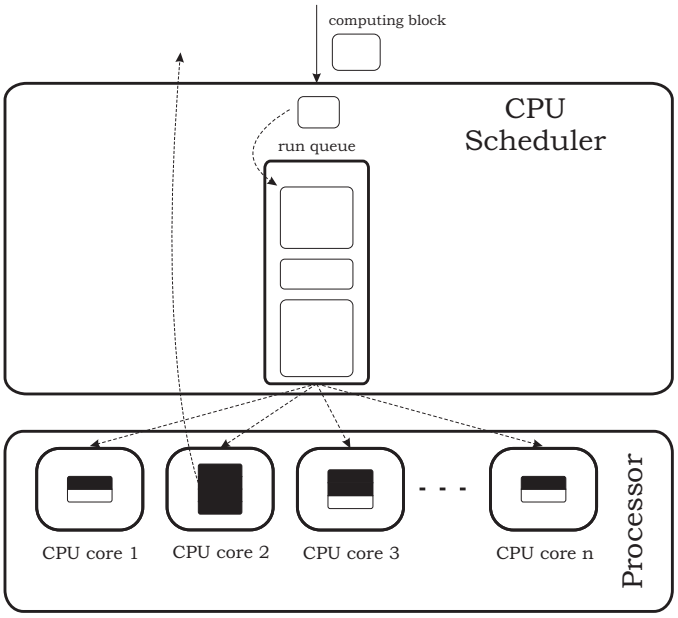


Figure 3.7: Example of a CPU scheduler using a FIFO strategy in SIMCAN

Algorithm 1 Modeling of a CPU scheduler in SIMCAN using a Round-Robin strategy

Require: processor P, run queue q

// Is there a new incoming block in the CPU system?
1: **if** (newIncomingBlockArrives()) **then**
2: $Cb \leftarrow$ divideBlockInIPTportions (Cb, P_{MIPS}, P_{tick})
3: $queue \leftarrow insert(queue, Cb)$
4: **end if**
// Assigning the corresponding computing blocks to idle cores
5: **for all** $((index)$ such that $(CPUcore_{index} \in P))$ **do**
6: **if** $isIdle(CPUcore_{index})$ **then**
7: **if** $(!queue.isEmpty())$ **then**
8: $CPUcore_{index} \leftarrow$ send $(queue.getFirstBlock(), CPUcore_{index})$
9: **end if**
10: **else**
11: $Cb_{index} \leftarrow getBlock(CPUcore_{index})$
12: **if** $(time_{remaining}(Cb_{index}) == 0)$ **then**
13: sendBackToApp (Cb_{index})
14: **if** $(!queue.isEmpty())$ **then**
15: $CPUcore_{index} \leftarrow send(queue.getFirstBlock(), CPUcore_{index})$
16: **end if**
17: **else**
18: **if** (isCurrentQuantumComplete(Cb_{index})) **then**
19: $queue \leftarrow insert(queue, Cb_{index})$
20: send $(queue.getFirstBlock(), CPUcore_{index})$
21: **end if**
22: **end if**
23: **end if**
24: **end for**

3.3.1.3 CPU Scheduler in SIMCAN using a priorities strategy

Another strategy for modeling a CPU scheduler in SIMCAN consists on assigning priorities to each computing block. Thus, the priority of a computing block Cb_a is noted such that $\rho(Cb_a)$.

Those priorities are assigned to each computing block that arrives to the computing system, and the CPU core is allocated to the computing block with the highest priority. When several computing blocks (Cb_a and Cb_b) have the same priority ($\rho(Cb_a) = \rho(Cb_b)$), those computing blocks are scheduled following a FIFO policy (see section 3.3.1.2).

The strategy for assigning priorities to each computing block varies for each system. For each tick executed in a CPU core, all priorities of the computing blocks located in the run queue are updated. If a new computing block Cb_c is placed in the run queue, then it is inserted by keeping sorted all computing blocks by their priorities. Thus, if there is currently executing a computing block Cb_a in $CPUcore_k$ and Cb_c is placed in the run queue with higher priority than Cb_a, such that $(\rho(Cb_c) > \rho(Cb_a))$, then Cb_a will be forced to abandon $CPUcore_k$ and Cb_c will start its execution.

Figure 3.8 shows an example of a CPU scheduler using a priorities strategy in SIM-

CAN. In this example, a new computing block arrives to the computing system. That block is inserted in the run queue by keeping sorted all computing blocks located in this queue. In this figure, each block has an associated priority beside it.

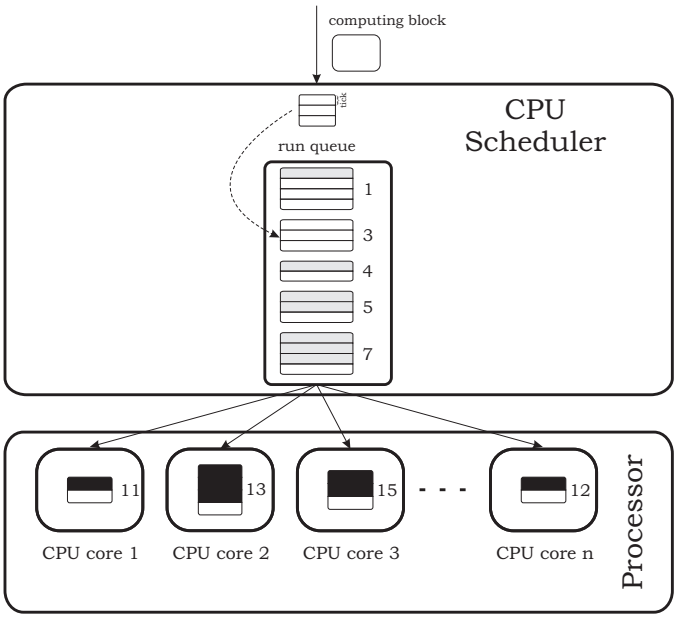


Figure 3.8: Example of a CPU scheduler using a priorities strategy in SIMCAN

Algorithm 2 shows the algorithm for modeling a CPU scheduler using a priorities strategy in SIMCAN.

3.3.2 Strategies for modeling the memory system

In SIMCAN, the memory system has been modeled using two different components. First component called physic memory is in charge of simulating the physical characteristics of the memory. Those characteristics include mainly the size of the memory space and latency times. The second component called memory manager is in charge of managing the memory accesses. Currently, this component manages the memory accesses needed for applications and disk cache. However, future works of this thesis include extend this functionality for supporting virtual memory.

Therefore, the memory can be modeled as a finite set of contiguous pages such that:

$$M = \{Mp_1, Mp_2, Mp_3, ..., Mp_{numPages(M)}\}$$

where the size of a memory page Mp that belongs to the memory M is noted as $size_{Mp}(M)$. Although the size of the memory page used is fully customizable, in most cases the value used for this is 4 KB, as occurs in the Linux Operating System (section 8.1, [BC05]). The total size of the memory space is noted as $memSize(M)$. Then, the number of pages in memory can be calculated using the equation 3.4.

$$numPages(M) = \frac{memSize(M)}{size_{Mp}(M)} \tag{3.4}$$

Algorithm 2 Modeling of a CPU scheduler in SIMCAN using a priorities strategy

Require: processor P, run queue q

// Is there a new incoming block in the CPU system?
1: **if** (newIncomingBlockArrives()) **then**
2: $Cb \leftarrow$ divideBlockInIPTportions (Cb, P_{MIPS}, P_{tick})
3: $queue \leftarrow insertSortedByPriority(queue, Cb)$
4: **end if**
// Assigning the corresponding computing blocks to idle cores
5: **for all** $((index)$ such that $(CPUcore_{index} \in P))$ **do**
6: **if** $isIdle(CPUcore_{index})$ **then**
7: **if** $(!queue.isEmpty())$ **then**
8: $CPUcore_{index} \leftarrow$ send $(queue.getFirstBlock(), CPUcore_{index})$
9: **end if**
10: **else**
11: $Cb_{index} \leftarrow getBlock(CPUcore_{index})$
12: $Cb_{first} \leftarrow queue.getFirstBlock()$
13: **if** $(\rho(Cb_{first}) > \rho(Cb_{index}))$ **then**
14: **if** $(time_{remaining}(Cb_{index}) == 0)$ **then**
15: sendBackToApp (Cb_{index})
16: **else**
17: $queue \leftarrow insertSortedByPriority(queue, Cb_{index})$
18: **end if**
19: $CPUcore_{index} \leftarrow$ send $(Cb_{first}, CPUcore_{index})$
20: **else**
21: **if** $(time_{remaining}(Cb_{index}) == 0)$ **then**
22: sendBackToApp (Cb_{index})
23: **end if**
24: $CPUcore_{index} \leftarrow$ send $(queue.getFirstBlock(), CPUcore_{index})$
25: **end if**
26: **end if**
27: **end for**

The main tasks performed by the memory manager are:

- Managing memory accesses from applications.
- Allocating memory pages required by applications.
- Freeing memory pages requested by applications.
- Managing a cache system for disk data.

In SIMCAN, the memory system divides the memory space for two different purposes: memory used for application space and memory used for disk cache space. Figure 3.9 shows the basic schema for modeling the memory system in SIMCAN and each one of its regions.

Memory used for application space can be defined as a finite set of memory pages, such that:

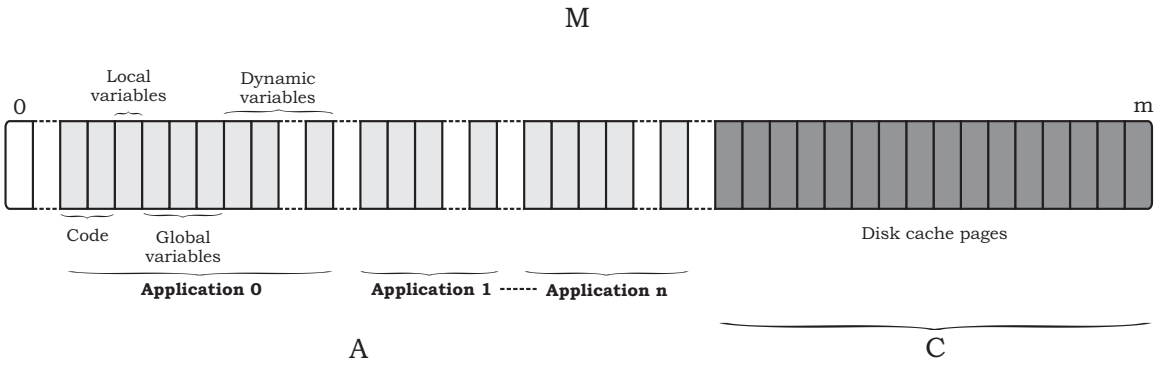


Figure 3.9: Basic schema for modeling the memory system in SIMCAN

$$A = \{Ap_1, Ap_2, Ap_3, ..., Ap_x\} | \forall i (0 < i \leq x), (0 < x \leq (|M| - |C|)), Ap_i \in M$$

This memory is allocated in a dynamic memory area, which means that this memory space can grow and shrink depending on the allocating and freeing operations invoked by applications. Otherwise, memory used for disk cache space can be defined as a finite set of memory pages such that:

$$C = \{Cb_1, Cb_2, Cb_3, ..., Cb_{numCachePages}\} \forall i (0 < i \leq numCachePages), Cb_i \in M$$

where $numCachePages$ is the number of pages used for allocate disk data. This parameter can be customized, but it cannot be modified during the execution of the simulation. Therefore, in some cases, cache disk space will contain empty blocks that will be not used for any purpose.

The set of free memory pages of a given memory M can be defined as:

$$freeMemPages(M) = M - (A \cup C)$$

Similarly, the set of pages currently used in a memory M is defined as:

$$M_{used} = \{A, C\} | \forall A, C, \in M (A \cap C = \emptyset)$$

3.3.2.1 Memory space used for applications

Each application divides its memory space in a set of different areas. The purpose of each one of those areas is to allocate:

1. Code.
2. Global variables.
   2.1 With initial value.
   2.2 Without initial value.

3. Local variables.

4. Dynamic variables.

5. File projection

6. Shared memory.

Currently in the SIMCAN simulation platform the first four memory areas can be modeled and simulated. Otherwise, file projection and shared memory are not currently supported in this simulation platform, whereof those features will be included in future works.

The amount of memory needed for allocating both code and global variables, can be configured in the application to be modeled by setting those parameters with the corresponding values. Those values will not change during the execution of the application. Thus, the size of those areas of memory can be calculated easily using the executable file of the application to be modeled. Then, it is responsibility of the user deciding whether those amounts of memory are important to be simulated in the corresponding environment. Therefore, although SIMCAN supports modeling those areas of memory, users may choose whether this feature must be enabled or not.

Modeling the memory used for local variables is more complex than modeling the previous memory areas, because this memory area is dynamic, whereof this area can grow and shrink depending on the variables used. In SIMCAN, this amount of space is not calculated by the simulation platform itself. Instead, each application has to be modified for calculating exactly the required amount of memory space for those purposes.

In this case, the application to be modeled must include two hints for each function invoked by that application. First hint must be included at the beginning of each invoked function, and the other hint must be included at the end of that function. Basically, this hint is a function implemented in the core of the simulation platform, which contains the amount of memory needed for allocating local variables used in each invoked function. Thus, first hint will allocate the required amount of memory, and second hint will free the same amount of memory previously allocated.

Although this memory is not mandatory to perform simulations, it can be calculated if the user considers that this amount of memory could be a factor with an important impact on the overall system performance.

The space of memory used for dynamic variables is calculated using the functions provided by API of the memory system (see listing 3.2). Each application in SIMCAN can allocate and free the required memory pages for its own use. Those amounts of memory are checked by the memory manager and treated accordingly. This component is in charge of assigning the requested amount of memory requested by an application, or deny this request if the amount of free memory is less than the memory required by the application. Similarly, this component is in charge to free the amounts of memory requested by applications, updating the number of free memory pages in the system.

It is important to mention that all data that belongs to the same area of memory, is stored in a whole number of pages. Thus, a page of memory cannot contain data that belongs to different areas of memory. Furthermore, this system can be very useful for

calculating the amount of memory requested for each application, which is very important to analyze the amount of memory needed when scaling up applications.

3.3.2.2 Memory space used for disk cache

Disk cache is a mechanism that allows the system to keep in memory some data that is normally stored on a disk. Thus, further accesses to that data can be satisfied quickly without accessing the disk.

Disk caches are crucial for system performance, because repeated accesses to the same disk data are quite common. An application that interacts with data stored in a disk usually is entitled to access repeatedly to read or write the same disk data.

When an application requests a data block from disk, the memory manager first checks if that block is already stored in one of the cache pages. If the block is not already in the cache space, a new page is added to the cache and filled with the data read from the disk. If there is enough free memory pages, the block is kept in a cache page for an indefinite period of time and can then be reused by other applications without accessing the disk. Similarly, before writing a block of data to a block device, the memory manager verifies whether the corresponding block is already included in the cache space; if not, a new page is added to the cache memory and filled with the data to be written on disk. The I/O data transfer does not start immediately, but it is delayed giving a chance to the applications to further modify the data to be written. This method is also called, write-through.

The disk cache can be modeled using several levels of detail. For example, a module that simulates quickly the memory system with low detail, uses a basic formula for calculating statistically if a block is stored in memory or not. Then, the corresponding latency time is applied for calculating the amount of time needed for accessing this block.

Otherwise, disk cache can be modeled with a high level of detail. For instance, using a module that contains a list of blocks for simulating the blocks that are in memory. Moreover, blocks stored in memory are managed using two different lists for read and write requests. In this module, several algorithms for managing blocks can be used, like the read-ahead and write-through algorithm. Algorithm 3 shows the algorithm for read data form disk using cache algorithms. Similarly, Algorithm 4 shows the algorithm for write data to disk using cache algorithms.

The benefits of using a very detailed module for this purpose are two-fold. First, the accuracy obtained is greater than using a module with a basic formula for mimic the behavior of the memory system. Second, an exhaustive analysis of the behavior of the memory can be done. The drawback of using high level of detail is that it requires more CPU power than a simpler module for simulating the memory system.

3.3.3 Strategies for modeling the storage system

Storage system performance is one of the major concerns that arise on large computing networks. The I/O system is usually a system bottleneck in most of the computing systems [PGK88]. Detecting the cause of the problem could be an easy task on a single computer or a small network, but detecting the problems and their causes in large computing networks is not a trivial task. Using simulation environments can help to solve it out.

Algorithm 3 Disk cache algorithm for reading data

Require: Application *App* requests x bytes from *disk*.
Ensure: x bytes in a variable stored in application space.

```
    // Split application request in blocks
 1: numBlocks = x \ size_Mb(M)
 2: AR ← {ARb_1, ARb_2, ARb_3, ..., ARb_numBlocks}
    // For each block (AR_b ∈ AR), search in cache when idle
 3: for all ((block) such that (block ∈ AR)) do
 4:   while isBusy(M) do
 5:     block()
 6:   end while
 7:   if (block ∈ C) then
 8:     C ← updateList(C, block)
 9:     delay ← calculateDelay()
        // If current block is NOT in cache
10:   else
11:     block ← read(disk, block)
        // Is there at least one free block in cache?
12:     if (numFreeBlocks(C) > 0) then
13:       C ← insert(C, block)
          // All blocks in cache are used
14:     else
15:       C ← removeBlock(C)
16:       C ← insert(C, block)
17:     end if
18:   end if
19:   wait(delay)
20:   send (block, App)
21: end for
```

In SIMCAN, the storage system has been modeled using a set of components that consists of:

- I/O redirector.
- File system.
- Volume manager.
- Disk drives.

I/O redirector, file system and volume manager represent the software part of the storage system. Otherwise, disk drives represent the hardware part.

The I/O redirector is a module in charge of redirecting file requests to the corresponding file system, which could be local or remote.

File system is the more complex piece of the whole SIMCAN simulation platform. Basically, file system is in charge of translating data requests from applications to a list of

Algorithm 4 Disk cache algorithm for writting data

Require: Application *App* writes x bytes to *disk*.
Ensure: x bytes written to disk.

```
    // Split application request in blocks
 1: numBlocks = x \ size_Mb(M)
 2: AR ← {ARb_1, ARb_2, ARb_3, ..., ARb_numBlocks}
    // For each block (AR_b ∈ AR), search in cache when idle
 3: for all ((block) such that (block ∈ AR)) do
 4:   while isBusy(M) do
 5:     block()
 6:   end while
      // If current block is in cache
 7:   if (block ∈ C) then
 8:     block ← setDirty(C, block)
 9:     delay ← calculateDelay()
        // If current block is NOT in cache
10:   else
11:     if (numFreeBlocks(C) > 0) then
12:       C ← insert(C, block)
          // All blocks in cache are used
13:     else
14:       writeDirtyBlocks(C, disk)
15:       C ← updateCacheList(C)
16:       if (numFreeBlocks(C) > 0) then
17:         C ← removeBlock(C)
18:       end if
19:       C ← insert(C, block)
20:     end if
21:   end if
22:   wait(delay)
23:   send (block, App)
24: end for
```

blocks which contains the requested data. For simulating file systems, we propose to use statistical models of the data block distribution that is performed by the file system to be simulated. Moreover, in section 3.3.3.2 is described the process for obtaining the statistical models of two well-known general purpose file systems.

Volume manager is in charge of managing the data block operations sent from file system. This component is described in detail in section 3.3.3.3

Finally, disk drive is in charge of storing data blocks. Basically this component calculates the amount of time spent for processing the data block operations that came from volume manager. This component is described in detail in section 3.3.3.4.

Figure 3.10 shows the basic schema of a storage system modeled using SIMCAN. Due to the great importance of this system, it has been modeled targeting to simulate a wide variety of I/O configurations. Following is described a brief overview of the process, since

a file operation invoked by applications arrives to the storage system, until this operation is completely performed.

This process starts when applications executed in a simulated environment invoke a file operation. Those operations are provided by the API of the storage system which are described in detail in section 3.4. Once this operation arrives to the storage system, the I/O redirector selects the corresponding file system in charge of managing the requested file. Then, the file system translates this operation in a corresponding data structure that depends directly of the operation requested. For example, for read and write operations the file system translates those requests in a list of blocks that contains the data involved. Each file operation requested from applications can be defined as a pair that consists of the type of the requested operation and a set of parameters, such that:

$$app_{fileOp} = \{operation_{type}, params\}$$

where *app* is the application that invokes the file operation, $operation_{type}$ is the concrete operation to be performed, and *params* is the set of parameters needed for performing $operation_{type}$. Usually, those parameters consist of the file name involved in the operation, the offset in the file and the size of data to be processed.

In this example (see figure 3.10) an application invokes a read operation. This operation arrives to the storage system, where the I/O redirector examines the path of the involved file in the operation, in order to select the file system that contains such file. Then, the corresponding file system translates the data request to a list of blocks. Next, this list of blocks is sent to the volume manager.

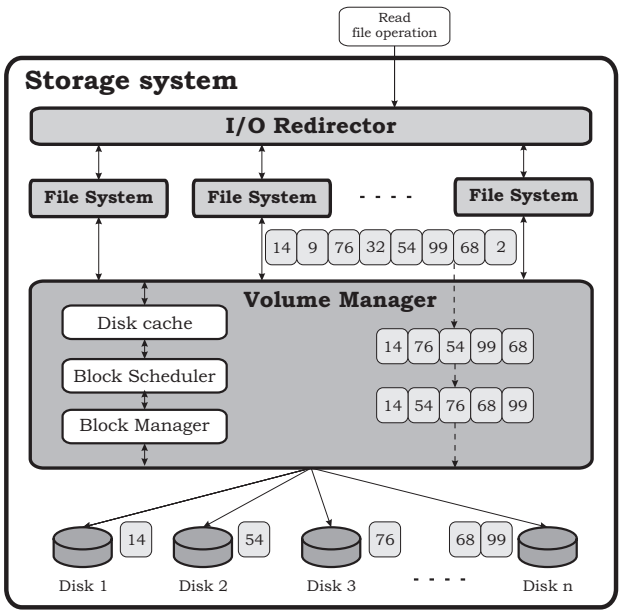


Figure 3.10: Basic schema for modeling the storage system in SIMCAN

When the list of blocks arrives to the volume manager, the disk cache calculates the blocks that are stored in this memory, and then a new list that contains only the blocks that are not stored in cache is generated. In this example, blocks number 9, 32 and 2 are

stored in cache. Then, a list containing blocks 14, 76, 54, 99 and 68 are sent to the block scheduler.

Next, the disk scheduler sorts this incoming list of blocks using the corresponding strategy. Finally, once the list of blocks has been processed, this list is sent to the block manager, which redirects each block request to the disk drive that contains the corresponding data.

3.3.3.1 The I/O redirector module

The main purpose of this component is to redirect each file request sent by applications, to the corresponding file system in charge of managing the file that contains such data.

This component let mount several file systems in a unique name space. The method for locating a file in a concrete file system is to analyze the full name of the file (including the directories and subdirectories). Therefore, remote file systems can be configured in simulated environments using the SIMCAN simulation platform.

Figure 3.11 shows an example of how a remote file system is modeled in SIMCAN. In this example there are two nodes: Node A and Node B. Node A contains two file systems, a local file system and a remote file system. All requests processed by the local file system will be sent to the local storage system. By the contrary, requests processed by the remote file system component will be sent to the file system hosted in Node B through the network system. Otherwise, Node B contains a local file system.

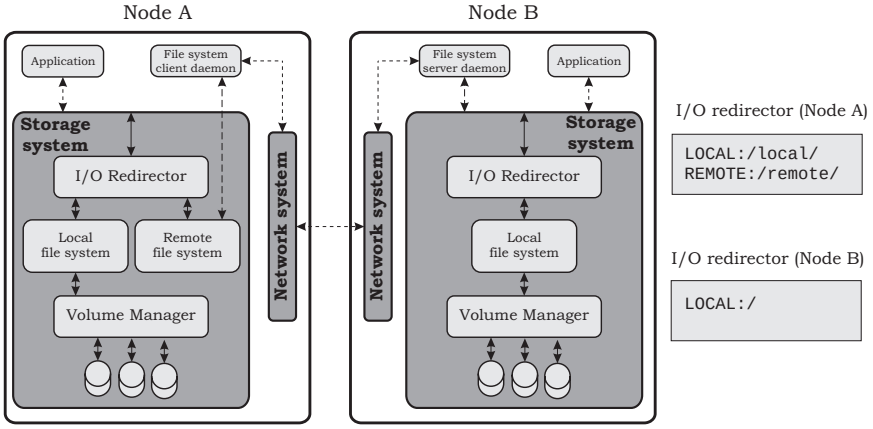


Figure 3.11: Example of a remote file system modeled in SIMCAN

The I/O redirector module is in charge of selecting the file system in charge of managing each request received in the storage system. This selection is based on the path of each involved file. Thus, the I/O redirector has to be configured by indicating the paths of the files contained in each file system. In this example, the configuration of the I/O redirector hosted in Node A contains two lines, first line is in charge of configuring the local file system, and second line is in charge of configuring the remote file system. Basically those lines indicate whether the file system is local or remote, and the corresponding mounting point. The I/O redirector hosted in Node B contains only one line because this node contains only one file system.

Moreover, file system daemons are needed in order to use remote file systems. Basically,

those daemons are applications that only can receive requests. For instance, client daemons receive requests from the remote file system. Then, those requests will be sent to the corresponding node that contains the server daemon. Otherwise, the server daemon receives requests from the client daemon in order to communicate such request to the corresponding storage system. The IP address of the node that contains the server daemons is configured in the client daemon component.

3.3.3.2 Modeling and simulating the file system

A file system is a piece of software in charge of organizing a set of files located in a storage device. Main tasks of file system include the followings:

- Managing the distribution of the data blocks on the disk.
- Mapping the requested data blocks into the corresponding disk blocks.
- Managing free blocks.
- Storing and managing file metadata information.

The main goal of this section is to obtain a statistic model of the data block distribution that is performed by the file system. Wherefore, a model for several general purpose file system layouts will be obtained.

Besides those tasks are very important, the two first tasks have more influence in the final service time. The weight of the data block distribution on the final service time is due to the fact that disks have different access times depending on which blocks are requested and in which order. The best way to include this effect in a simulation platform is to include a file system component that calculates the data block distribution. There are two different methods for performing this task:

- Implementing the file system functionality or, at least, the data block distribution.
- Using a statistic estimation to approximate the data block distribution.

The first method is less interesting due to the following reasons:

- It is difficult and complex to implement.
- It requires storing a great amount of metadata.
- It should be remade each time there was a file system change. In contrast, the second method only requires changing the statistic estimation function.

Nowadays there are several works using statistic estimations in obtaining file system characteristics, like [Mit03]. Most commonly they are used to obtain distributions of file sizes, which are used for creating benchmarks (benchmark for Web servers, for example). But there are few works done on obtaining data block distributions to simulate I/O systems.

Following, a formal definition of the components involved in the modeling of the file system will be presented.

A *disk* can be defined as a finite set of data blocks that are physically located along its surface, such that:

$$disk = \{Db_1, Db_2, ..., Db_n\} | (0 < n \leq \frac{disk_{size}}{disk_{blockSize}})$$

where the number of data blocks in a disk depends directly both on the size of the disk, noted as $disk_{size}$, and the size of the block used, noted as $disk_{blockSize}$.

Similarly, a $file$ can be defined as a set of data blocks such that:

$$file = \{Fb_1, Fb_2, ..., Fb_f\}$$

Those blocks represent the logic view of the file (see figure 3.12). Each one of those blocks is mapped to one block in the surface of the disk. Then, for a given file system Fs that manages the files stored in $disk$, exists a translation function ψ that is in charge of mapping the blocks of all files contained in $disk$, such that:

$$\psi(Fb_i, file_j, disk) = Db | Db \in disk, \forall j | file_j \in disk, \forall i | Fb_i \in file$$

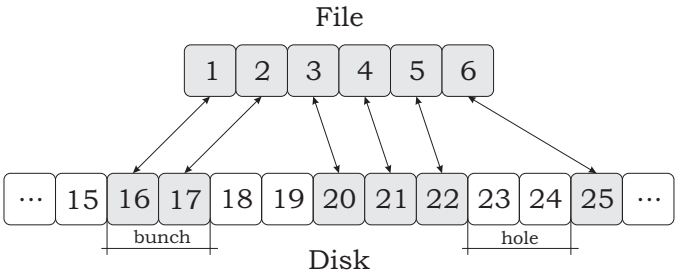


Figure 3.12: Data block distribution along the disk surface

The factors involved on the allocation distribution for a certain file are quite a few, but there are three of them with capital importance:

- Type of file system, because each one has its own strategies for managing free blocks and file distribution.
- The amount of free blocks remaining in the disk, because as the number of free blocks increases, the sizes of block group increase as well.
- The file size, because as the file size increases, the number of block groups increases as well, and also their size.

In order to obtain the pursued statistical distribution, a set of tasks must be performed:

- Obtaining different layouts for different file systems, like the ones used in the real world, that include all the important features that could affect the data block distribution.

- Extracting from those layouts the most relevant characteristics of the data block distributions.
- Modeling the behavior of those characteristics using statistical distributions.

Basically, there are two methods for obtaining those layouts. One of them is to mimic the file system layout used in real systems with the characteristics we are searching for. The problem is that a group of layouts with very specific features is needed (for example the disk usage ratio), which we have not found in the state-of-the-art. Furthermore, obtaining those real layouts is a very complex and difficult task.

Other authors propose a method to obtain those file system layouts by using statistical distributions. For instance, works like [DB99][Mit03] propose heavy-tailed distributions for modeling this kind of file system layouts. This is the approach that we have chosen because it is easier to handle and it is realistic enough to our purpose, as demonstrated in the works commented before.

A problem to have in mind is that the block distribution of a certain file is not independent of the block distribution of the other files. In fact, the data block distribution will evolve as the number of performed operations in the file system increases. This process is called the aging of a file system.

In order to extract the more relevant characteristics needed for performing the data block distribution model, this task will be focused on studying how the data blocks are grouped. Thus, a data block distribution of a given file can be modeled as a set of block groups (that we call block bunches) separated by a number of blocks that do not belongs to this file (we call this a hole).

Thus, a bunch of a given $file$ stored in $disk$ can be defined as a set of data blocks that belongs to $file$ and are stored contiguously in $disk$, such that:

$$bunch_{file}(disk) = \{Fb_1, Fb_2, ..., Fb_b\} | \forall i (0 < i \leq b) Fb_i \in file,$$

$$\forall j (0 < j < b) | Fb_j \in file, \psi(disk, file, Fb_j) - \psi(disk, file, Fb_{j+1})| = 1$$

Similarly, a hole of a given $file$ stored in $disk$ is defined as a set of blocks that does not belong to $file$, stored contiguously in $disk$ between two bunches of $file$, such that:

$$hole_{file}(disk) = \{Db_1, Db_2, ..., Db_h\} | \forall i (0 < i \leq h) \nexists Fb_j \in file, Db_i = \psi(j, file, disk),$$

$$\exists Fb_a, Fb_b \in file, \psi(Fb_a, file, disk) = Db_0, \psi(Fb_b, file, disk) = Db_{h+1}$$

Therefore, files with only one bunch are denominated contiguous files, and others that are composed by several block bunches separated by a hole are denominated non-contiguous files. This block distribution model let us estimate the service time by obtaining which are the head movements. In order to obtain this model, three parameters that should be obtained from the previous obtained file system layouts have been used. Those parameters are:

- Contiguous file: Determines if a file is composed by one or more block bunches.

- Block bunch size: Determines the size of the contiguous block groups of a given file.
- Distances between block bunches: Determines the distance between two consecutive block bunches of the same file.

A part of this approach consists on gathering the values of those three parameters from the file system layouts proposed before. The data obtained is used to estimate a statistical distribution that fits with this data.

The first step on every modeling process is to obtain a representative group of samples of the studied population. This population consists of files with its own block distribution. As described previously, the most important factors on the block distribution of a certain file are: the kind of file system used, the amount of free blocks remaining on disk, and the file size. A certain combination of these factors will imply a corresponding block distribution model.

In order to solve this problem, an approach based on clustering the population following these three factors has been used. Then, a set of samples of each cluster have been used for calculating the data block distribution of such cluster. Observing those three factors we can conclude that the kind of file system and the disk usage ratio are intrinsic values of any existing file system layout. Thus, in order to obtain different samples for those factors, a set of different layouts has been obtained. In contrast, the file size is intrinsic for each file. Thus, different samples for this factor in the same layout can be obtained. Having all these concepts in mind, the following strategy to obtain the samples has been used.

First of all, several layouts for each kind of all involved file systems and for all range of disk usage have been created. Those layouts correspond to a typical general purpose system. In order to accomplish this task, two general purpose file systems have been used (Ext2 [CTT94] and ReiserFS [rei07]). Then, the disk usage range has been split (0%...100%) in 10 slices of 10% each one.

Those layouts were created using the following technique. The process started with an empty hard disk drive. This drive was filled using several threads that create or destroy files along the time for aging the file system. The system had to ensure that the disk drive maintained a certain disk usage ratio during all the process.

The sizes of the files created in this process followed a statistic distribution that models a typical file system layout. Therefore, some distribution samples of file sizes stored in a server used in our department for many years have been extracted. Using these distribution samples, a heavy-tailed distribution has been estimated in order to simulate the obtained results for the used benchmark. Thus, a Weibull distribution with parameters α=0.29027 and β=43.23635 has been used.

Once the file system layouts have been obtained, then a concrete set of samples must be extracted. Each layout generates a set of samples for a concrete kind of file system and a concrete disk usage ratio. Thus, the generated samples from a certain layout will differ on the size of the files included. For each layout 5 samples were obtained, according to the file sizes (less than 10KB, 10KB-100KB, 100KB-1MB, 1MB-10MB and 10MB-100MB). Thus, each obtained sample is defined by:

- The kind of the file system used (Ext2 or ResiserFS).
- The disk allocation ratio (10 slices of 10% each one).

- The file size (less than 10KB, 10KB-100KB, 100KB-1MB, 1MB-10MB and 10MB-100MB).

Those samples with the same characteristics were used to generate a specific block distribution. All those block distributions were modeled as a list of grouped blocks. Thus, both the size of each group of blocks and the distance between them have been calculated. This approach consists on modeling those two values, the block bunch size and the distance between them, as random variables that follow a certain statistical distribution.

This approach has a problem that involves the last group of blocks. The size of those block groups does not follow the same distribution that the rest of the block groups. The reason is that the sizes of those groups are limited by the size of the whole file. Therefore, this group of blocks will be excluded from the block bunch size estimation.

A special case occurs when the file is composed by only one bunch. In this case, the size of those groups of blocks is the same that the file size. Thus, this kind of files should be treated separately. Then, the following data for each file in the sample have been collected:

- The file is contiguous or not.
- The size of the block bunches from the non-contiguous files (last bunch is discarded).
- The distance between two consecutive block bunches of the file.

Using the data previously collected, we propose a schema for reproducing the data block distribution of any file. This schema will use the model of block bunches separated with holes, but we also consider whether the file is contiguous or not. Algorithm 5 shows the algorithm that implements this schema.

Using this schema, it is simple to map a request for a certain file. Once the block distribution model of a given file has been calculated (composed by a list of block bunches and holes) the offset and request size can be used for obtaining a subset of the data block distribution corresponding with this request, which is also represented by a list of bunches and holes.

In order to estimate the parameters using random variables, a stochastic study of the obtained samples must be performed. Due to previously obtained samples which represent each one of the clusters of samples from the population has its own characteristics, a different model for each classification has been calculated. In order to accomplish this, different random variables for each sample model maintaining the original structure have been obtained. Thus, in order to obtain the data block distribution of a certain file, a cluster that fits the most with the characteristics of a file must be chosen. Then, concrete values from the parameters listed below are calculated.

- The parameter that shows whether a file is contiguous or not is modeled through a Bernoulli distribution, which will be true or false depending on the probability that this file is contiguously stored. This distribution requires a value that shows the probability of this file for being contiguous. In order to obtain this probability, a set of distribution samples is needed to calculate parameters of a Weibull distribution that fits the experimental results as much as possible.

Algorithm 5 Algorithm for modeling the data distribution of a file F in file system FS

Require: File F, file system FS, diskRatio
Ensure: List of bunches and holes for modeling file F

```
    // Establishing if File is contiguous or not
 1: if (isContiguous(FS_type, diskRatio) then
 2:    bunch[1] ← sizeOf(F)
 3:    distance[1] ← 0
       // Determining the bunches and holes
 4: else
 5:    i ← 0
 6:    fileRest ← sizeOf(F)
 7:    while (distance[i] ≠ 0) do
 8:       i ← i + 1
          // Obtaining bunch size
 9:       bunchAux ← trunc (sizeOfBunch (FS_type, sizeOf(F), diskRatio))
          // Obtaining hole size
10:       distAux ← trunc (distances (FS_type, sizeOf(F), diskRatio))
          // Determining if this is the last bunch
11:       if (fileRest > branchAux) then
12:          bunch[i] ← bunchAux
13:          distance[i] ← distAux
14:       else
15:          bunch[i] ← fileRest
16:          distance[i] ← 0
17:       end if
18:    end while
19: end if
```

- The parameter that estimates the size of block bunches is modeled using a Weibull distribution. This distribution has been chosen because its flexibility and because it is a heavy-tailed distribution. We assumed that if the file size fits a heavy-tailed distribution, then the size of the bunch belonging to a file will also fit a heavy-tailed distribution. This estimation is made using several sample distributions of the block bunches, discarding the last bunch of the file for the reasons explained before.

- The last parameter corresponds to the estimation of the distances between two consecutive block bunches of the same file. The corresponding sample distributions are used for calculating the parameters of a Weibull distribution.

Following, the tables with the α and β parameters of the resulting weibull distributions, obtained from the performed modeling study will be presented. Therefore, in order to demonstrate the accurary of such modeling study, several qq-plot charts will be shown. Each one of those qq-plot charts represents a specific cluster of the analyzed population, which means a concrete disk percentage occupancy, a concrete file size and a file system type. Due to the high number of qq-plot needed for covering all possible combinations of the obtained parameters, a concrete set of parameters have been chosen with the purpose

of creating a reduced set of qq-plot charts. Otherwise, showing all those charts is this works is impractical.

The quantile-quantile (q-q) plot is a graphical technique for determining if two data sets come from populations with a common distribution. A q-q plot is a probability plot, which is a graphical method for comparing two probability distributions by plotting their quantiles against each other. By a quantile, we mean the points taken at regular intervals from the cumulative distribution function (CDF) of a random variable. That is, the 0.3 (or 30%) quantile is the point at which 30% percent of the data fall below and 70% fall above that value. A 45-degree reference line is also plotted. If the two sets come from a population with the same distribution, the points should fall approximately along this reference line. The greater the departure from this reference line, the greater the evidence for the conclusion that the two data sets have come from populations with different distributions. Q-Q plots can also be used as a graphical means of estimating parameters in a location-scale family of distributions.

Table 3.1 and table 3.2 shows the α and β parameters of several Weibull distributions. These distributions are used to estimate the ratio of contiguous files for each one of the cluster of samples obtained from the population.

Figure 3.13 shows two qq-plot examples of the ratio of contiguous files distributions. Left chart shows the probability of a given file, between 10 and 100 KB, for being contiguous in the Extended 2 file system with a 70% of disk occupancy. In this chart we can appreciate that the weibull distributions fit very well due to almost all points fall near to the line. Right chart shows the same probability plot but using a Reiser file system. In this case we can appreciate that all points are concentrate in the interval near to 90%, otherwise to the extended 2 files system, where the points are distributed in a longer interval, between 84% and 98%.

Table 3.3 and table 3.4 shows the α and β parameters of several Weibull distributions. These distributions are used to estimate the size of the block bunches for each one of the cluster of samples obtained from the population.

The group of files which size is less than 10KB is treated such a special case because they could not get more than 3 blocks. That means that most of them are contiguous (in some cases all of them), and the rest shows a block bunch of 1 block (there is almost none with 2 blocks).

Figure 3.14 shows two qq-plot examples of the bunch size distributions. Left chart shows the bunch size distribution of a given file, between 1 and 10 MB, in the Extended 2 file system with a 90% of disk occupancy. Right chart shows the same distribution but using a Reiser file system. In those charts we can appreciate that weibull distribution fits very good due to the majority of points fall very close to the line. Both two charts are similar, with the exception that the Reiser file system is less accurate in the last quantile.

Table 3.5 and table 3.6 shows the α and β parameters of several Weibull distributions. These distributions are used to estimate the distance between consecutive block bunches for each one of the cluster of samples obtained from the population.

Figure 3.15 shows two qq-plot examples of the hole size distributions. Left chart shows the hole size distribution of a given file, between 100KB and 1 MB, in the Extended 2 file system with a 80% of disk occupancy. Right chart shows the same distribution but using a Reiser file system. In those charts we can appreciate that weibull distribution fits good due

to the majority of points fall very close to the line. The distribution used for the Reiser file system fits better than that used for the extended 2 file system. It can be appreciated due to the right chart contains more points closer to the line. However, this is the distribution that fits with less accuracy. This is caused because estimating the distribution of holes in a given file system is more complex that estimate the distribution of bunch sizes and the contiguous files.

File Size / Disk Ratio	<10KB	≥10KB <100KB	≥100KB <1MB	≥1MB <10MB	≥10MB <100MB
10%	All Contiguous	1,323230	11,38610	4,521457	All Non-Contiguous
		0,965934	0,703956	0,395348	
20%	56,309636	93,15731	20,52223	2,516753	All Non-Contiguous
	0,9814452	0,925358	0,463522	0,111411	
30%	56,309636	89,25030	10,91014	2,888433	All Non-Contiguous
	0,9814452	0,878312	0,302650	0,036821	
40%	73,154356	82,95295	13,54341	2,481393	All Non-Contiguous
	0,9804823	0,872349	0,276824	0,024934	
50%	46,044541	68,50647	11,63993	1,468665	All Non-Contiguous
	0,9637632	0,819155	0,177393	0,010984	
60%	52,076648	79,17988	11,31144	1,437263	All Non-Contiguous
	0,9428502	0,794753	0,183740	0,012724	
70%	32,175196	76,52483	7,997440	1,307311	All Non-Contiguous
	0,9381173	0,754909	0,140887	0,006558	
80%	28,758123	69,13898	9,235985	1,591022	All Non-Contiguous
	0,9214031	0,729568	0,137616	0,010662	
90%	22,160365	49,40721	10,99656	1,223993	All Non-Contiguous
	0,9048285	0,683374	0,109472	0,005700	
100%	5,5846902	5,175926	2,713411	0,876507	All Non-Contiguous
	0,6766179	0,377974	0,040299	0,001406	

Table 3.1: β and α Weibull distribution values to estimate the contiguous probability parameter for Ext2 FS

The obtained results are very accurate in most cases. The contiguous and bunch size parameters fit extraordinarily well with the assumptions made. In contrast, the distances between bunches (holes), have a few glitches in the fitness of the distribution.

3.3.3.2.1 Strategies for modeling parallel file systems

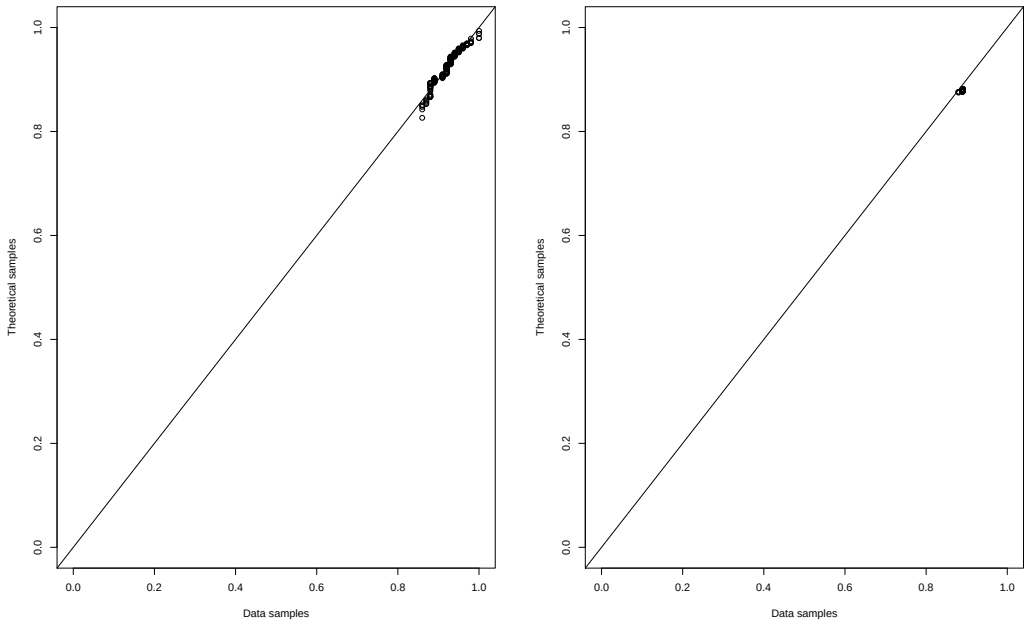
Parallel file systems are a kind of file system that consist of a set of server nodes, usually called I/O nodes, which contains data that can be accessed in parallel from the rest of the nodes. Parallelism in this kind of file systems is obtained using several independent I/O nodes supporting one or more secondary storage devices. Then, data are striped along those nodes in order to allow both parallel access to different files, and parallel access to the same file. If several I/O nodes are used in parallel, the performance can be increased in two ways:

1. Allowing parallel access to different files by using several disks and servers.

2. Striping data using distributed partitions, allowing parallel access to the data of the same file.

File Size / Disk Ratio	<10KB	≥10KB <100KB	≥100KB <1MB	≥1MB <10MB	≥10MB <100MB
10%	All Contiguous	619,9321	39,70583	39,70583	1,636751
		0,980758	0,804779	0,804779	0,057951
20%	255,6710	1684,414	34,59164	18,54875	3,025447
	0,988078	0,966321	0,725775	0,166024	0,009228
30%	433,2875	185,1901	176,4333	13,30076	1,267067
	0,987247	0,952328	0,655343	0,135573	0,007676
40%	177,1228	560,5762	56,34171	41,24978	All Non-Contiguous
	0,981840	0,932879	0,571159	0,072701	
50%	85,55648	83,65626	14,77872	6,259649	All Non-Contiguous
	0,973341	0,910064	0,514969	0,057206	
60%	351,9228	88,78054	44,82858	17,63940	All Non-Contiguous
	0,970490	0,897011	0,491902	0,054396	
70%	474,6164	732,9433	30,12113	29,39341	All Non-Contiguous
	0,964336	0,884296	0,459703	0,046842	
80%	54,81594	70,43497	55,08419	62,00382	All Non-Contiguous
	0,946731	0,855953	0,397066	0,030608	
90%	78,56610	48,94079	8,909514	2,409420	All Non-Contiguous
	0,938696	0,802132	0,301086	0,011775	
100%	7,250649	3,066549	0,502074	0,350998	All Non-Contiguous
	0,587863	0,265726	0,005236	6,61E-05	

Table 3.2: β and α Weibull distribution values to estimate the contiguous probability parameter for ReiserFS



(a) Ext2 file system

(b) Reiser file system

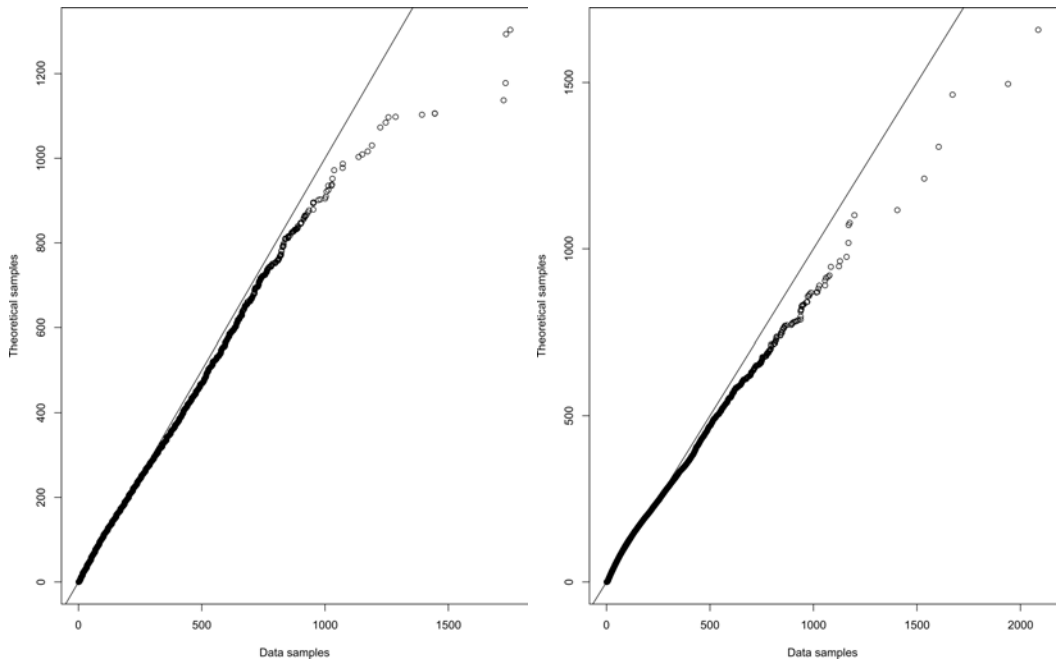
Figure 3.13: qq-plot of contiguous probability for files between 10KB-100KB and 70% disk ratio

Disk Ratio \ File Size	<10KB	≥10KB <100KB	≥100KB <1MB	≥1MB <10MB	≥10MB <100MB
10%	All Contiguous	0,996990	1,282864	0,910406	0,685902
		5,671231	43,44072	183,0593	591,0184
20%	Value=1	1,146203	1,100370	0,778886	0,625856
	Always	5,065225	34,11654	121,1688	280,4090
30%	Value=1	1,137328	0,989241	0,727632	0,568342
	Always	5,078811	28,13313	87,23370	168,9680
40%	Value=1	1,112851	0,940452	0,694574	0,558184
	Always	4,543842	25,10682	67,47077	104,6392
50%	Value=1	1,140626	0,896141	0,655388	0,520011
	Always	4,501042	20,43478	46,98559	65,16852
60%	Value=1	1,102315	0,887035	0,661330	0,506361
	Always	4,165466	20,21413	46,79699	58,88015
70%	Value=1	1,096969	0,865733	0,646279	0,505339
	Always	3,958488	18,57642	40,13278	49,11946
80%	Value=1	1,063725	0,839464	0,624326	0,477668
	Always	3,736526	16,29773	32,63591	35,74111
90%	Value=1	0,984623	0,797547	0,625309	0,506969
	Always	3,199165	13,45244	25,60777	28,06224
100%	Value=1	0,855544	0,650916	0,594232	0,619796
	Always	1,240227	1,355010	1,347538	1,782166

Table 3.3: β and α Weibull distribution values to estimate the block bunch size parameter for Ext2 FS

Disk Ratio \ File Size	<10KB	≥10KB <100KB	≥100KB <1MB	≥1MB <10MB	≥10MB <100MB
10%	All Contiguous	2,271130	1,341484	0,717478	0,617078
		8,272070	28,79251	101,3308	525,5712
20%	Value=1	1,447095	0,996795	0,688844	0,617705
	Always	6,280666	22,28983	71,09551	180,3048
30%	Value=1	1,294811	0,883728	0,643573	0,512333
	Always	5,351929	16,75546	40,19987	77,83894
40%	Value=1	1,211942	0,852019	0,607304	0,480736
	Always	4,786548	13,86539	27,19116	55,32873
50%	Value=1	1,159276	0,810269	0,567433	0,452386
	Always	4,256116	10,83368	17,75029	35,64011
60%	Value=1	1,089191	0,771463	0,541470	0,475683
	Always	3,785614	9,135287	13,94413	34,83233
70%	Value=1	1,060923	0,763049	0,511560	0,435247
	Always	3,488661	7,864873	9,916556	23,72846
80%	Value=1	1,042575	0,720645	0,474600	0,446431
	Always	3,237344	6,560203	7,211891	23,54883
90%	Value=1	1,007913	0,692531	0,491174	0,416680
	Always	2,974623	5,625142	7,008675	16,35779
100%	Value=1	0,879038	0,553750	0,364548	0,456650
	Always	1,612414	1,774211	1,154953	9,143521

Table 3.4: β and α Weibull distribution values to estimate the block bunch size parameter for ReiserFS



(a) Ext2 file system (b) Reiser file system

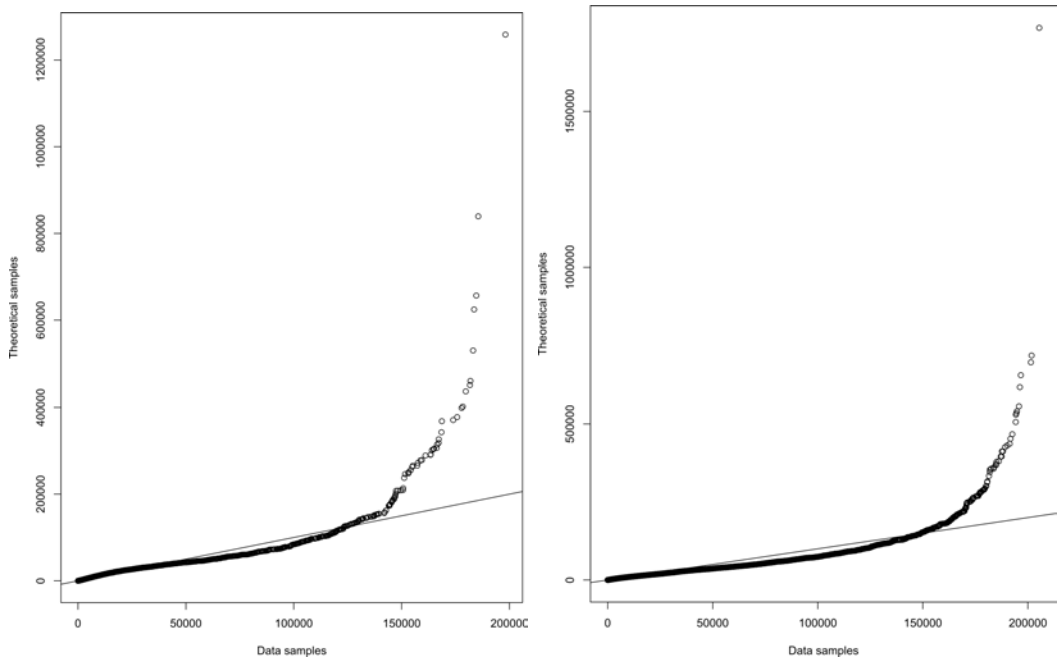
Figure 3.14: qq-plot of block bunch size for files between 1MB-10MB and 90% disk ratio

Disk Ratio \ File Size	<10KB	≥10KB <100KB	≥100KB <1MB	≥1MB <10MB	≥10MB <100MB
10%	All Contiguous	0,653165	0,638266	0,608935	0,539024
		633,8339	616,6030	490,4353	449,1463
20%	0,343761	0,422329	0,504942	0,486182	0,498149
	678,4843	550,4278	423,3293	354,2766	326,2738
30%	0,435833	0,404815	0,435754	0,405207	0,449747
	175,1264	465,2363	365,9841	231,7518	227,6115
40%	0,546782	0,348905	0,377654	0,366203	0,410490
	323,4131	277,2571	260,9681	175,4221	168,8293
50%	0,419907	0,314219	0,327776	0,318404	0,338830
	424,9047	182,6201	173,8922	103,9476	95,35894
60%	0,345741	0,306321	0,326415	0,310567	0,308154
	548,3225	142,6138	158,6823	98,08962	69,65236
70%	0,412868	0,293287	0,290762	0,302492	0,280730
	151,9049	121,6979	119,9672	85,97720	46,70743
80%	0,362910	0,271796	0,262321	0,252131	0,230786
	175,1245	89,86200	80,41816	41,87846	18,71869
90%	0,348596	0,259798	0,258100	0,260208	0,189908
	137,9986	58,69267	58,56193	41,28546	5,245876
100%	0,206614	0,168983	0,105956	0,100930	0,100930
	17,43968	4,819231	0,049371	0,007181	0,007181

Table 3.5: β and α Weibull distribution values to estimate the distance parameter for Ext2 FS

Disk Ratio \ File Size	<10KB	≥10KB <100KB	≥100KB <1MB	≥1MB <10MB	≥10MB <100MB
10%	All Contiguous	0,401424	0,362564	0,427239	0,422948
		260,6537	194,0777	387,8600	332,7708
20%	0,333993	0,378566	0,402702	0,417329	0,436037
	370,7879	575,0266	401,8286	391,4378	333,1901
30%	0,407860	0,355484	0,362939	0,346291	0,368326
	1816,309	571,9133	346,8861	244,0743	241,1934
40%	0,334488	0,343485	0,336907	0,326827	0,334488
	195,2870	495,2299	270,4904	203,4703	195,2870
50%	0,353134	0,307061	0,299610	0,310089	0,304895
	504,8168	26,17534	190,6189	155,5280	136,6857
60%	0,373188	0,296201	0,290331	0,306730	0,332365
	378,0507	301,7020	163,3964	143,6213	169,6638
70%	0,311830	0,277591	0,281472	0,281871	0,283232
	203,5536	207,7682	119,5978	102,4723	101,1337
80%	0,289872	0,267046	0,270168	0,232007	0,323985
	174,0229	155,4032	102,9376	42,49982	155,6052
90%	0,251919	0,258602	0,264365	0,272833	0,251919
	59,44370	144,4898	89,18861	80,44415	59,44370
100%	0,251238	0,203370	0,247354	0,119940	0,303418
	34,33798	19,18014	40,56325	0,175040	79,19211

Table 3.6: β and α Weibull distribution values to estimate the distance parameter for ReiserFS



(a) Ext2 file system

(b) Reiser file system

Figure 3.15: qq-plot of distances between bunches for files between 100KB-1MB and 80% disk ratio

The main advantages provided by parallel file systems include a global name space, scalability, and the capability to distribute large files across multiple nodes. In a distributed environment, large files are shared across multiple nodes, making a parallel file system well suited for storage systems. Some examples of parallel file systems found in literature are: Vesta [CF96], PVFS [CIRT00], GPFS [GPF09], and Expand [CCC+03].

SIMCAN provides a general schema for modeling a generic parallel file system. This schema is showed in figure 3.16. Although this figure shows only one computing node, models of parallel file systems can support a high number of computing nodes. Basically, the idea is to provide a generic customizable schema, with the purpose of that each user can model the required parallel file system with a concrete behavior.

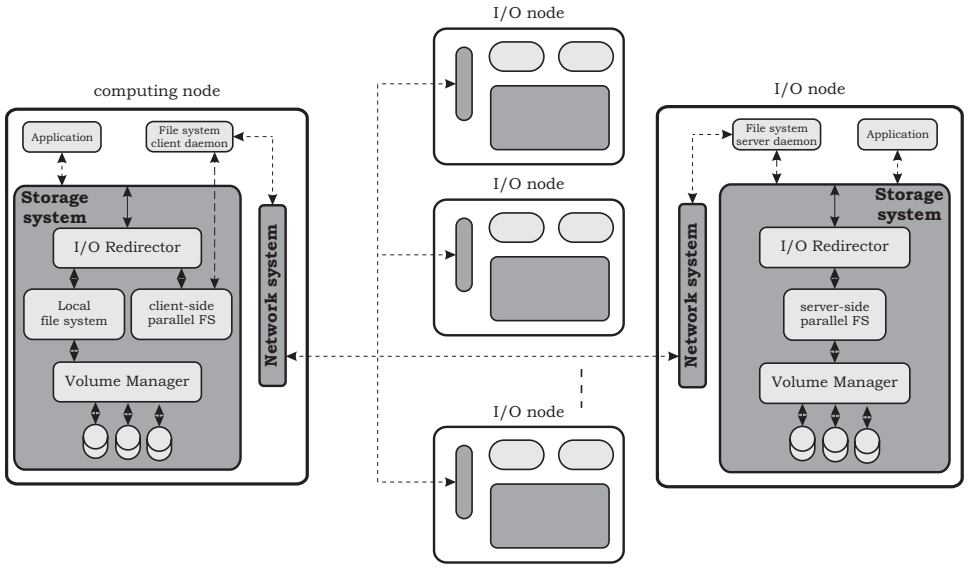


Figure 3.16: Basic schema for modeling a parallel file system in SIMCAN

This schema consists of next set of components, which can be programmed in order to simulate a concrete parallel file system:

- Client-side parallel FS. This component is in charge of locating the I/O nodes that contains the requested data. Then, when this component receives requests from applications, those requests are processed and sent to the corresponding I/O nodes.
- File system client daemon. This component receives requests from the client-side parallel FS. The objective of this component is to send those requests to the corresponding I/O node through the communication network.
- File system server daemon. This component receives requests from the file system client daemon. The objective of this module is to send the received requests to the local file system component, in this case the server-side parallel FS.
- Server-side parallel FS. This component is in charge of accessing data in the local node where it is hosted. This component receives requests from file system server daemon, which are performed in the local storage system.

This schema of a parallel file system involves two kinds of nodes: a set of nodes in charge of storing data, called I/O nodes, and a set of nodes that access to such data, called computing nodes. Thus, I/O nodes contain server-side file system and file system server daemon. Otherwise, computing nodes contain client-side file system and file system client daemon. Both daemons are used with the purpose to provide a homogeneous method for using the complete API provided by the simulation platform. Then, those daemons can be customized without making difficult its programming.

Currently, SIMCAN implements a basic parallel file system model. This model is targeted to build a unified file space using a set of I/O nodes for storing data. Thus, each file in this file space is partitioned in slices, which are distributed among a set of I/O nodes. Then, the objective is to maintain all files distributed in several I/O nodes in order to allow the access to several parts of each file in parallel. This parallelism is obtained when the parts to be accessed are stored in different I/O nodes. The number of slices of a given file depends directly of the stride size used for partitioning such file, which is totally customizable. Thus, both parallelism for accessing parts of the same file, and accessing parts of different files can be achieved.

The distribution of files along the I/O nodes is transparent for applications. Also, the algorithm used for distributing the slices of each file depends of each system. This proposed model uses a hash function in order to calculate the I/O node that contains the first slice of a given file, and then using a Round-Robin algorithm for locating next slices. The main objective of the hash function is to provide the maximum level of distribution among I/O nodes, because the greater level of distribution, the greater probability in order to obtain parallelism.

A hash function that provides a good distribution consists on using the name of the corresponding file involved in the data request. This function is used in the Expand file system [CCC+03], which is used for calculating the I/O node that stores the first slice of a given file. Formula 3.5 shows this function.

$$firstNode = \sum_{i=1}^{i=strlen(fileName)} fileName[i] \bmod numIOnodes \tag{3.5}$$

where $fileName[i]$ is the i-th character of $fileName$, and $numIOnodes$ is the total number of I/O nodes. Thus, the algorithm used in the client-side parallel FS for translating data requests in the corresponding set of requests sent to the server-side file systems, is showed in algorithm 6.

This provided model of a parallel file system uses general purpose file systems, like Ext2 and Reiser FS, for modeling the server-side file system component. Therefore, this component is modeled using the techniques described in section 3.3.3.2. Moreover, this model does not use metadata because each slice of any file can be calculated using both the previously described function and the Round-Robin algorithm.

3.3.3.3 Volume Manager

Volume manager is a software component in charge of performing read and write operations of data blocks, sent from file systems. Basically, this component receives data block requests from file system and it must locate the requested data for performing the corresponding

Algorithm 6 Algorithm for locating each slice of a given file request

Require: $stripeSize, numIOnodes, fileName, offset$, and $dataSize$.
Ensure: A set of requests to the corresponding I/O nodes.

```
    // Calculates the first slice of current request
1:  firstSlice ← offset / stripeSize
    // Calculates where is stored the first slice of current request
2:  firstIOnode ← (firstNode(fileName) + firstSlice) mod numIOnodes
    // Initializes local variables
3:  dataRemaining ← dataSize
4:  currentSlice ← firstSlice
5:  currentNode ← firstIOnode
    // For each slice involved in current request
6:  while (dataRemaining > 0) do
7:      sendRequest(IONode_currentNode, currentSlice)
        // Update variables for next request
8:      dataRemaining ← (dataRemaining − stripeSize)
9:      currentSlice ← currentSlice + 1
10:     currentNode ← ((currentNode + 1) mod numIONodes)
11: end while
```

operation. In SIMCAN, a volume manager is modeled using three components: data block cache, data block scheduler and block manager.

The data block cache is a component in charge of storing disk data blocks in a cache memory. This memory manages disk data blocks, at opposite of the memory described in section 3.3.2.2, which is in charge of managing file data blocks. In most systems, usually only one of those memories is enabled. However, in this simulation platform those memories are fully customizable, and the user is in charge of configuring them for simulating the corresponding architecture.

The model of this memory is the same that the disk cache described in section 3.3.2.2. Therefore, algorithms 3 and 4 are used in this component for accomplish its purpose. The main objective of this component is to maintain in memory the most frequently used disk data blocks managed by the volume manager. Then, those blocks requested from the file system that are stored in this cache, will be processed in this module, performing much faster the incoming requests from file system. The rest of blocks of incoming requests will be send to the data block scheduler. However, this cache can be enabled or disabled, depending of the requirements of the modeled architecture. In the case of disabling this component, all requests will pass through directly to the data block scheduler.

Second component in the volume manager is the data block scheduler. The main purpose of this component is to schedule the data block requests that are not stored in the data block cache. Depending of the modeled architecture, this component can use the appropriate strategies for scheduling data block requests, like FIFO, elevator algorithm, and CSCAN. However, this module can also be enabled or disabled depending on the requirements of the modeled architecture. As occurs with the data block cache, if this component is disabled, then all incoming requests will pass through directly to the next component, which in this case is the block manager.

Finally, the block manager is in charge of redirecting the incoming data block requests to the disk that contains such data. This module can be configured for modeling a wide range of architectures. For example, if the storage system contains a single disk drive, this component will send directly all requests to that disk. When several disks are used, this component can be configured for using a set of disk drives as a single volume. Moreover, this module can be also configured to act like a RAID system. For example, in a RAID level 1 (Mirroring) this module calculates the disks where each block is stored and the corresponding mirror disk. At opposite of the block cache and block scheduler modules, this module cannot be disabled.

3.3.3.4 Disk drives

In SIMCAN a disk is the component in charge of calculating the amount of time needed for reading and writing data blocks. The definition of how a disk drive is modeled in SIMCAN was previously described in section 3.3.3.2.

In this thesis, three different approaches are described for modeling and simulating disks in the SIMCAN simulation platform.

First approach consists on using a fixed value for the bandwidth of read and write operations, which are measured in MB/s. Then, given a disk d with read bandwidth d_{read}, the amount of time (in seconds) needed for read x bytes is calculated using equation 3.6.

$$time_{read}(disk, x) = \frac{x}{disk_{read} \cdot 10^6} \tag{3.6}$$

Similarly, given a disk d with write bandwith d_{write}, the amount of time (in seconds) needed for write x bytes is calculated using equation 3.7.

$$time_{write}(disk, x) = \frac{x}{disk_{write} \cdot 10^6} \tag{3.7}$$

The main problem of this method is the poor accuracy obtained, because this method does not take into account the position of each block along the disk surface, but the number of bytes to be processed. The main advantage of this method is the high speed of simulation and the simplicity to be implemented.

The second approach consists on using the linear interpolation technique for calculating the access times of each data request. Basically, the idea of this approach is to execute a benchmark for gathering relevant information of the disk to be modeled, by calculating access times using a set of pre-defined parameters. Then, using the linear interpolation technique with that information gathered from the disk, each disk request can be calculated.

The information obtained from the benchmark consists of a set of access times calculated using 2 parameters: the size of the data requested in the current operation, and the distance (measured in number of disk blocks) between the last block processed in the previous operation and the first block processed in the current operation. Thus, the benchmark is executed twice. In the first execution, the benchmark calculates access times for read operations. Similarly, in the second execution the benchmark calculates access times for write operations.

Due to calculating the access times for all possible combinations of data size and distance between processed data is impractical, a set of pre-defined sizes for those parameters are used. Basically, those sizes goes from 512 bytes to 1 Gigabyte, using sizes multiple of 2, such that: 512 bytes, 1 Kilobyte, 2 Kilobytes, 4 Kilobytes, ..., 1 Gigabyte.

Therefore, if a disk request match with the values used for calculating the corresponding access times, the response is immediate because this result has been previously calculated by the benchmark. Otherwise, the linear interpolation technique, the information obtained by the benchmark and the parameters of current operation are used for calculating the access time of such operation.

Algorithm 7 shows the algorithm of the benchmark executed for gathering the needed information of the disk to be modeled. Once this benchmark is executed, algorithm 8 is used for calculating the time spent on performing a disk request.

The linear interpolation, showed in figure 3.17 can be calculated using equation 3.8.

$$li(x, x_1, x_2, y_1, y_2) = y = y_1 + (x - x_1) \cdot \frac{y_2 - y_1}{x_2 - x_1} \tag{3.8}$$

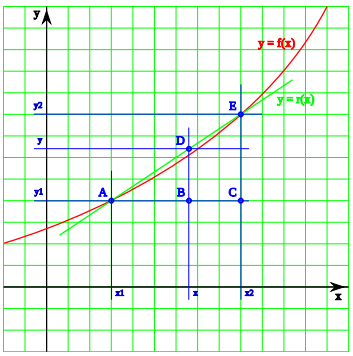


Figure 3.17: Linear interpolation chart

Finally, last approach consists on using a diskSim model. DiskSim [BSSG08] is a disk simulator which models with a very high level of detail the behavior of disk drives. Then, a disk model is included in the SIMCAN simulation platform for simulating the corresponding disk model. Moreover, solid state disks can be simulated using diskSim.

3.3.4 Strategies for modeling the network system

In SIMCAN, the network system let the nodes of distributing systems to interchange data through a communication network. Usually, this communication network consists of a set of communication devices and network links. Then, the topology of a distributed model can be defined as a non-directed graph $G = \{V, E\}$ such that V is a finite set that contains the nodes and communication devices in the distributed system, and E represent the link between two components that are connected in the model. For modeling the network system, two different approaches are described in this section.

First approach consists on calculating the time spent for sending a message msg of $size$ bytes from node $node_A$ to node $node_B$. This calculation is made based on the path crossed by msg from $Node_A$ to $Node_B$, which involves a set of communication links and

a set of communication devices. For calculating this path, the Dijkstra's shortest path algorithm is used [Dij59]. Then, applying this algorithm, the shortest path for sending a message from $node_A$ to $node_B$, both contained in G will generate a graph G' that contains a set of vertex $V' \in V$ and a set of edges $E' \in E$, such that:

$$Dijkstra(node_A, node_B, G) = G'$$

Each link $\in E$ has associated two features that characterizes such communication link in the network: latency (measured in μs) and bandwidth (measured in MB/s). Similarly, each component $\in V$ has associated a latency or processing time (measured in μs) that is the amount of time spent by a component in the model to figure out where to forward a data unit. Those features are configured by the user in the simulated environment, which can be obtained from the public features of real components. Then, the time spent for sending a message msg of $size$ bytes from node $node_A$ to node $node_B$ can be calculated using equation 3.9.

$$time = \sum_{i=0}^{|V'|-1} V'i_{CPU} \cdot 10^{-6} + \sum_{i=0}^{|E'|-1} E'i_{latency} + \frac{msg_{size}}{E'i_{bandwidth}} \cdot 10^{6} \tag{3.9}$$

where $V'i_{CPU}$ is the time needed for processing the message in each communication device reached by msg, $E'i_{latency}$ is the latency time for each link crossed by msg, msg_{size} is the size of the message sent, and $E'i_{bandwidth}$ is the bandwidth of each link crossed by msg.

The advantage of using this approach is performance, because only few calculations are needed for estimating the time for a message to be sent from one node to another. The disadvantage of using this approach is the poor accuracy obtained. One of the main causes of this lack of accuracy is that congestions produced in the network are not calculated. This occurs because this method does not consider the amount of messages that are currently being sent through the network, but the time needed for sending each message from a source node to a destination node.

The second approach for simulating the network is to use the INET framework. This method is more complex and detailed. This framework contains modules for simulating completely a network system, including network protocols like TPC or UPD. The main advantage of this method is the high level of accuracy obtained, because all elements that compose a network are simulated. However, the main drawback is performance, because this high level of detail needs a considerably CPU power to be calculated.

The INET framework provides a set of modules such as routers, switches and network protocols, for building a wide range of networks. Thus users can build several network architecture models like LAN (Local Area Networks) and WAN (Wide Area Network). Moreover, those networks can be configured as wired and wireless networks.

In order to ease the development of new applications in SIMCAN, a set of calls for managing remote connections has been included into the API module in SIMCAN. Thus, a great asset of this module is that the simulated application has not to know the method used for simulating the network, due to it is transparent to the application, because the same API is used for connecting each simulated application with the network system.

Algorithm 7 Benchmark used for gathering information of the disk to be modeled

Require: Disk drive *disk* to be modeled

Ensure: Two-dimensional array with the access times for read and write operation in *disk*

// Initializing

1: MAX_SIZES ← 23
2: sizes[0] ← distances[0] ← 0
3: sizes[1] ← distances[1] ← 512

// Building array with data sizes and distances

4: **for** (i=2; i<MAX_SIZES; i++) **do**
5: sizes[i] ← sizes[i-1] * 2
6: distances[i] ← distances[i-1] * 2
7: **end for**

// Get access times for read operations

8: **for all** ($currentSize \in sizes$) **do**
9: **for all** ($currentDistance \in distances$) **do**
10: $offset \leftarrow random(0, sizeOf(disk))$
11: $time_{total} \leftarrow 0$
12: $iteration \leftarrow 0$
13: **while** ($iteration < MAX\_ITERATIONS$) **do**
14: $time_{start} \leftarrow getTime()$
15: $read(disk, offset, currentSize)$
16: $time_{end} \leftarrow getTime()$
17: $offset \leftarrow offset + currentDistance$
18: $iteration \leftarrow iteration + 1$
19: $time_{total} \leftarrow time_{total} + (time_{end} - time_{start})$
20: **end while**
21: $times_{read}[currentSize_{index}][currentDistance_{index}] \leftarrow \frac{time_{total}}{MAX\_ITERATIONS}$
22: **end for**
23: **end for**

// Get access times for write operations

24: **for all** ($currentSize \in sizes$) **do**
25: **for all** ($currentDistance \in distances$) **do**
26: $offset \leftarrow random(0, sizeOf(disk))$
27: $time_{total} \leftarrow 0$
28: $iteration \leftarrow 0$
29: **while** ($iteration < MAX_ITERATIONS$) **do**
30: $time_{start} \leftarrow getTime()$
31: $write(disk, offset, currentSize)$
32: $time_{end} \leftarrow getTime()$
33: $offset \leftarrow offset + currentDistance$
34: $iteration \leftarrow iteration + 1$
35: $time_{total} \leftarrow time_{total} + (time_{end} - time_{start})$
36: **end while**
37: $times_{write}[currentSize_{index}][currentDistance_{index}] \leftarrow \frac{time_{total}}{MAX\_ITERATIONS}$
38: **end for**
39: **end for**

Algorithm 8 Calculates access time to disk using linear interpolation

Require: $operation, data_{size}, data_{offset}, timeArray_{read}, timeArray_{write}$
Ensure: $time_{req}$

// Initializing
1: MAX_SIZES ← 23
2: sizes[0] ← distances[0] ← 0
3: sizes[1] ← distances[1] ← 512
// Building array with data sizes and distances
4: **for** (i=2; i<MAX_SIZES; i++) **do**
5: sizes[i] ← sizes[i-1] * 2
6: distances[i] ← distances[i-1] * 2
7: **end for**

// Get higher and lower indexes for the data request in the size array
8: $index_{x1} \leftarrow getLowerIndexForSize(sizes, data_{size})$
9: $index_{x2} \leftarrow getHigherIndexForSize(sizes, data_{size})$

// Get higher and lower indexes for the data request in the distance array
10: $index_{y1} \leftarrow getLowerIndexForDistance(distances, data_{offset})$
11: $index_{y2} \leftarrow getHigherIndexForDistance(distances, data_{offset})$

// Calculating linear interpolation for lower index values
12: $auxResult_a \leftarrow li(data_{offset}, distances[index_{y1}], distances[index_{y2}],$
$timeArray_{operation}[index_{x1}][index_{y1}], timeArray_{operation}[index_{x1}][index_{y2}])$

// Calculating linear interpolation for higher index values
13: $auxResult_b \leftarrow li\ (data_{offset}, distances[index_{y1}], distances[index_{y2}],$
$timeArray_{operation}[index_{x2}][index_{y1}], timeArray_{operation}[index_{x2}][index_{y2}])$

// Calculating linear interpolation for obtaining the access time for current data request
14: $time_{req} \leftarrow li(data_{size}, sizes[index_{x1}], sizes[index_{x2}], auxResult_a, auxResult_b)$

3.4 SIMCAN API module

The API module plays a very important role in the component that simulates an operating system. Basically this module contains a set of system calls which are offered as an API (Application Programming Interface) for all applications executed in a SIMCAN node. A system call is the mechanism used by applications to request a service from the operating system. Thus, those system calls provide the interface between applications and the operating system (see figure 3.3). Moreover, researchers can write applications to be simulated in SIMCAN using this API.

This API is totally implemented inside the API module. Also, this API determines the vocabulary and calling conventions that the applications should employ to use the operating system services, including the specification of the corresponding interfaces.

In order to maintain a certain degree of compatibility, this API pretends to be a subset of POSIX. POSIX is the name of a family of related standards specified by the IEEE, which defines an API that allows a wide range of common computing functions to be written, such that they may operate on many different systems. The API provided by SIMCAN through the API module is shown in listings 3.1, 3.2, 3.3, and 3.4.

Following, grouped by system, all operations provided by the API module in SIMCAN are listed:

Listing 3.1: Functions provided by the API of the computing system

```
1 void simcan_request_cpu (long int numInstructions);
```

Basically the computing system manages a list of all received request from the applications and deliver each one to the corresponding CPU. The CPU can contain one or more CPU cores. Thus, all computing units are controlled by this system.

Listing 3.1 offers an interface for using the CPU service. Basically applications which request CPU processing must invoke the *simcan_request_cpu* function and specify the corresponding amount of instructions to be executed, measured in MIs (Million Instructions).

Listing 3.2: Functions provided by the API of the memory system

```
1 void simcan_request_allocMemory (int memorySize, int region);
2 void simcan_request_freeMemory (int memorySize, int region);
```

The main task of the memory system is to assign the corresponding amount of memory to each application that requires it. Thus, this module receives requests for memory allocation and calculates where and how this memory has to be assigned. This feature is very useful for analyzing the amount of memory used for each application, especially in large distributed environments.

In order to interact with the memory system, applications must use the interface specified in listing 3.2. Basically this interface consists of two functions. First function, called *simcan_request_allocMemory*, is in charge of allocating memory. Second function, called *simcan_request_freeMemory*, is in charge of freeing the previous allocated memory. Parameter *memorySize* indicates the amount of memory required (measured in bytes) to perform the corresponding operation. Parameter *region* indicates the region of memory involved in this operation (see section 3.3.2).

Listing 3.3: Functions provided by the API of the storage system

```
1 void simcan_request_open (char* fileName);
2 void simcan_request_close (char* fileName);
3 void simcan_request_create (char* fileName);
4 void simcan_request_delete (char* fileName);
5 void* simcan_request_read (char* fileName, unsigned int offset, unsigned int size);
6 void* simcan_request_write (char* fileName, unsigned int offset, unsigned int size)
      ;
```

The storage system is in charge of managing all accesses to data. The set of functions showed in listing 3.3 offers an interface to interact with the storage system, which basically consists of a set of functions for managing files.

Functions *simcan_request_open* and *simcan_request_close* are in charge of opening and closing respectively a file given its name.

Functions *simcan_request_create* and *simcan_request_delete* are in charge of creating and removing respectively a file given its name.

Finally, functions *simcan_request_read* and *simcan_request_write* are in charge of reading and writing respectively data in the file specified in the parameter called *fileName*. The amount of data to be read or written is specified in the parameter *size*. Finally, the parameter *offset* specifies the starting point in the file where requested data is processed.

Listing 3.4: Functions provided by the API of the network system

```
1 void simcan_request_createListenConnection (int localPort, string type);
2 void simcan_request_createConnection (string destAddress, int destPort, int id,
      string type);
3 void simcan_request_sendDataToNetwork (SIMCAN_Message *sm, int id);
4 void simcan_request_receiveDataFromNetwork (SIMCAN_Message *sm, int id);
```

The network system is in charge of managing connections with other applications located in remote nodes, and also processing both the received and sent packets. The network system API is showed in listing 3.4. Using this interface, applications can manage connections with remote nodes. Moreover, those applications are able to send and receive data through the network.

Function *simcan_request_createListenConnection* creates an incomming connection. Otherwise, function *simcan_request_createConnection* establishes a connection with a remote application. For instance, applications that act like a server will use the first function, and client applications will invoke the latter function in order to establish the corresponding connections.

In order to send and receive data, functions *simcan_request_sendDataToNetwork* and *simcan_request_receiveDataFromNetwork* must be used respectively. First function is in charge of sending data from the application that invokes this function to a remote application specified in the message *sm*. Second function is in charge of receiving data from a remote application specified in the message *sm*.

Besides functions provided by this API, SIMCAN also provides a high level layer for developing distributed applications. This layer is placed in the application component and provides standard interfaces for executing distributed applications. Currently SIMCAN has implemented an interface for executing MPI applications.

3.5 Incerasing the functionality of SIMCAN

One of the main design objectives of SIMCAN is to create an open simulation platform, which can increase its functionality by adding new components and simulators to its repository. Therefore, as the repository of SIMCAN becomes larger, the coverage of the simulated architectures using SIMCAN increases as well. This task can be done using two different methods:

- Adding new components to the repository of SIMCAN.
- Adding new or existing simulators to the repository of SIMCAN.

In this section, those methods are described in detail.

3.5.1 Developing new components in SIMCAN

The current version of SIMCAN provides a wide set of developed components in order to build Simulated Environments. Although using those components a great variety of architectures can be modeled and simulated, the design of this simulation platform let users add its own new components to the repository of SIMCAN. Thus, new environments with more specific configurations can be built.

Usually, adding new components to a simulation platform entails some constraints. As occurs in real world, new added modules have to respect predefined interfaces because this is the only way that let new modules to communicate with the existing ones.

The basic structure of SIMCAN is a set of modules that send and receive messages among themselves. Those modules send and receive messages across several predefined connections configured between such modules. Those connections set up a network that simulates the desired architecture. Initially, if two modules are connected through a link, those modules can interchange messages.

Those message objects have attributes, which some of them are used by the simulation kernel, and the rest are provided just for the convenience of the simulation programmer. Figure 3.18 shows the Message class hierarchy used in SIMCAN. Currently, there are four concrete message classes, where each class is used for the corresponding system:

- SIMCAN_CPU_Message is used in the computing system.
- SIMCAN_Memory_Message is used in the memory system.
- SIMCAN_IO_Message is used in the storage system.
- SIMCAN_NET_Message is used in the network system.

All those classes inherit from the parent class SIMCAN_Message. Moreover, each message class has a list of operations associated. Those operations match with the functions provided with the corresponding API (see listings 3.1, 3.3, 3.4, and 3.2).

Therefore, when a new component is developed, there are basically 3 methods for managing messages:

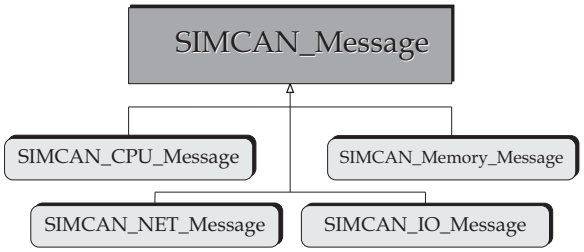


Figure 3.18: SIMCAN message hierarchy

1. Using one of the existing message types shown in figure 3.18.
2. Creating an specific message class for one of the four basic systems in SIMCAN, which inherits from one of the four message class system shown in figure 3.18.
3. Creating a new message type which has to inherit from the parent class.

The ability of a simulation platform to model such systems is constrained by a number of factors, including the capacity to accommodate large system models. In this case, those constraints consist on the way different components interchange messages. In other words, the component's interface. Thus, each component contains an interface that is perfectly defined by the message types and the list of operations that the corresponding component is able to manage.

Each system has its own interface (CPU, Storage, Memory, and Network) thus, each component that requires performing a request to a concrete system must respect the corresponding interface.

Figure 3.19 shows an example where several components interchange messages. Solid-lined boxes represent components and broken-lined boxes represent the interface of such components. Those interfaces contain both the message types that the component is able to manage and the corresponding list of operations. This example shows that an application sends four messages to the API module. Message_1 contains an open_connection operation, Message_2 contains a read_data operation, Message_3 contains a processing operation and Message_4 contains an allocating memory operation.

API module can receive four types of messages, one for each system previously described (see figure 3.18). Otherwise, all messages sent by the applications are the higher at the hierarchy (SIMCAN_Message). When the API receives a message, it makes the corresponding cast in order to send that message to the corresponding system.

Then, once Message_1 arrives to API module, a corresponding cast will be made to the class SIMCAN_NET_Message and will send this message to Network Service system. Following, API module performs a cast of message Message_2 to SIMCAN_IO_Message, and then this new message is sent to the storage system. The same goes to Message_3 and Message_4, when those messages arrive to API module, this module performs the corresponding casts to SIMCAN_CPU_Message and SIMCAN_Memory_Message respectively.

In summary, if a new component fulfils the corresponding interface, this new component will fit perfectly in the simulation platform. Those new components have to follow

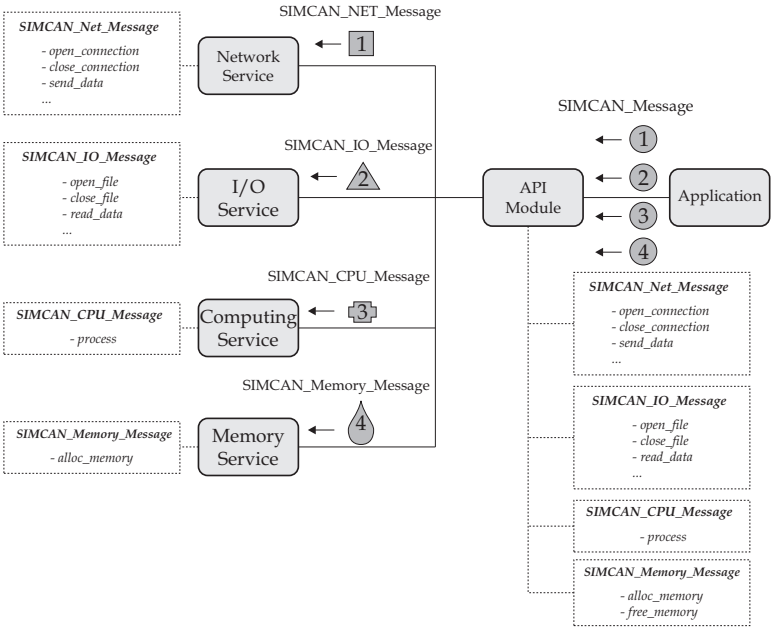


Figure 3.19: Example of module interface

several rules in order to be adapted well in the simulation platform. Those rules are the component interface, which is defined by the list of messages and operations this component can send and receive.

3.5.2 Adding new simulators to the SIMCAN framework

In this section is described the process for using an existing simulator to simulate a corresponding component. The main advantage of using existing simulators is that researchers do not have to care about coding the logic of the simulation, because that is provided by the simulator itself. Then simulators used in simulated environments require to be communicated with SIMCAN. This communication is performed using a wrapper.

A wrapper is a module, placed between the simulator and the components that need to use the simulator, which is in charge of establishing a communication between them. Thus, each request received in the wrapper can be processed and translated to invoke the corresponding function in the simulator. Figure 3.20 shows the process since the message request arrives to the wrapper module, until the response message is sent back.

First, a component sends a request message to the wrapper (1). This message has to be processed and translated in order to invoke the corresponding function in the simulator (2). Then, the corresponding function in the simulator library is invoked (3). When the simulator has processed the request, a result value is returned (4). This result value is processed and translated to a response message (5). Finally, this response message is sent back to the component that performed the initial request (6).

Figure 3.21 shows how a new simulator can be added to the SIMCAN repository. Basically there are two methods: using a static library or using a dynamic library.

Using a static library (see figure 3.21.a) requires compiling the entire framework. Then,

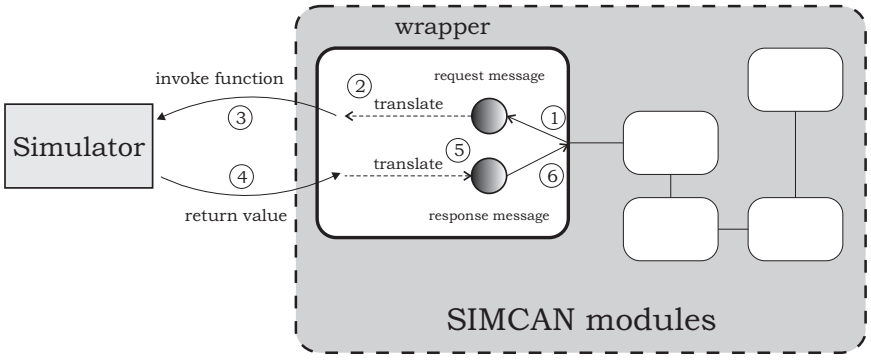


Figure 3.20: Example of a wrapper behavior that interacts with an external simulator

functions provided by the simulator can be invoked from every component in the repository of SIMCAN. The second method consists on using a dynamic library (see figure 3.21.b), which will be included by the wrapper module in order to invoke the functions provided by the simulator.

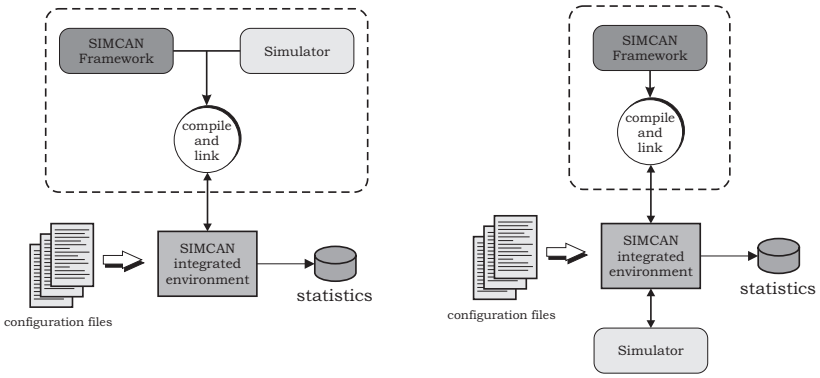


Figure 3.21: Methods for adding a new simulator to the repository of SIMCAN

3.6 Summary

In this chapter has been presented a proposal of a new simulation platform for creating simulation models that represent both real distributed system and applications.

The main contribution of this proposal is the possibility of modeling large systems by defining and configuring each one of the four basic systems independently. Therefore, users can focus the simulation in the parts they consider important and relevant for the corresponding research. Thus, large simulation models can be executed faster than simulating a complete system with a very high level of detail.

Finally, several methods for increasing the functionality of this simulation platform are proposed. Basically those methods consist on adding new components and simulators to the repository of the simulation platform.

Chapter 4

Modeling and Simulating computer architectures in SIMCAN

Creating simulated environments for distributed systems used to be a tedious and time-consuming task. Basically this process consists on writing configuration files that contains both the definition of the components involved in the simulation, and a list of parameters that characterizes each one for a specific purpose.

Moreover, the greater the size of the system to be modeled, the higher the complexity for modeling it. Due to this, creating those kinds of simulated environments requires a considerable amount of time and effort to be accomplished. The difficulty increase when adapting the simulation to be executed in a parallel environment because the components of the simulation must be grouped into parallel partitions as balanced as possible.

In this chapter, the process for creating simulated environments in SIMCAN is described in detail. Moreover, some proposed strategies for both ease this process and automatically accomplish the parallelization of those kinds of simulated environments are presented. Finally, the usefulness of those strategies by modeling real environments and showing their performance is presented.

4.1 Configuring simulated environments in SIMCAN

Usually, the task of modeling a simulated environment consists on defining and configuring a set of modules that represent components of a concrete real system. Depending on the configuration of each component, the overall system behavior will be modeled to a concrete purpose.

In SIMCAN, the definition and configuration of a modeled environment is divided in three different levels, each one with a corresponding level of abstraction. Thus, the greater level of configuration, the higher level of detail required in order to accomplish the configuration of such level. At least, configuring level 1 and level 2 are mandatory in order to define a simulated environment. Otherwise, configuration corresponding to level 3 are only used for modeling and simulating complex environment configurations, which is only needed in some environments that requires a high level of detail and customization.

Following, those three levels of configuration are described:

- Level 1 consists of the definition of the system corresponding to the environment to be modeled. Basically this level contains a list of all components, with their corresponding connections, involved in the environment to be modeled.
- Level 2 consists of the configuration of each component defined in level 1. This configuration basically consists in assigning values to the set of parameters of each component, in order to customize the behavior of the simulated environment.
- Level 3 consists on additional configuration files for customizing specific components that require it. In most cases those files are used in components that model a system with high level of detail, like parallel file systems.

Level 1 represents the higher level of abstraction (lower detail) in the model. Essentially, this level contains the description of the system to be modeled, which includes a set of components involved in the system and the connections between them. This description is specified in a plain text file using the NED language, which facilitates the modular definition of a distributed system. The channels and modules of a given system description can be reused in another model. Moreover, NED files can be loaded dynamically into simulation programs, or translated into C++ by the NED compiler and linked into the simulation executable.

Level 2 represent the customization of the level 1, which basically consists on customizing the behavior of each component in the model. As occurs with level 1, this level is completely defined using one plain text file where each parameter of all components in the model is configured. Parameters may be used to customize the behavior of each module, and to parameterize the model topology.

Parameters can take string, numeric or boolean values, or can contain XML data trees. Numeric values include expressions using other parameters and calling C functions, random variables from different distributions, and values input interactively by the user. Numeric-valued parameters can be used to construct topologies in a flexible way. Within a compound module, parameters can define the number of sub-modules, number of gates, and the way the internal connections are made. The number of parameters used to configure each environment gives the level of detail of the corresponding environment. Thus, the more detailed the model is, the higher number of parameters is needed for configuring each model and more accurate will be the simulation.

Finally, level 3 represents the higher level of detail in the model. Although this level is not required for all models, it can be very useful for modeling complex and very detailed systems. This level is oriented toward customization of specific architecture configurations, which require a high level of detail to be modeled. At this moment, the SIMCAN simulation platform provides only three components that require this level of configuration, which are following described:

- Servers list file. This file contains a set of IP addresses that belongs to concrete servers existing in the model. The objective of this file is to enumerate a set of servers that can be accessed from applications, with the purpose to establish a connection between the applications that load this file, and the corresponding server existing in this file.
- File system contents. This file contains a list of file names which will be pre-loaded in the corresponding file system. There are so many cases which we need an initial

state of the file system. Using this file, a file system can pre-load a list of files before starting the simulation. In this file are indicated the name of each file and its initial size.

- I/O redirector file. This file sets up a set of file system partitions. Those partitions can be local or remote. Thus, when a local request arrives to the I/O redirector module, this module calculates whether the request belongs to a local file system or, by the contrary, to a remote file system. This information is loaded from this file by I/O redirector module.

4.1.1 Proof case: Example of a basic distributed model using SIMCAN

In order to show how an environment can be modeled in the SIMCAN simulation platform, a basic simulated environment has been built. The characteristics of the simulated environment are listed in table 4.1 and table 4.2. Basically this environment consists of 2 computing nodes, 2 storage nodes and one switch. Those nodes are connected to the switch through an Ethernet Gigabit network. Figure 4.1 shows the simulated environment described in table 4.1 and table 4.2.

Computing nodes	**Storage nodes**
Ethernet Gigabit interface	Ethernet Gigabit interface
Dual-core processor	Tetra-core processor
Hard Disk of 400 GB	RAID storage system with 4 disks of 400 GB
A trace player application	
Ext2 File System (local FS)	Extended 2 File System (local FS)
NFS File system (mounted partition)	NFS server
1 GB of RAM memory	8 GB of RAM memory
Latency = 4 *us*	Latency = 4 *us*
Flushtime = 5 sg	Flushtime = 5 sg
Blocksize = 4 KB	Blocksize = 4 KB
2 read-ahead blocks	2 read-ahead blocks

Table 4.1: Characteristics of computing and storage nodes

Network
Network bandwidth = 1 Gbps
Network delay = 125 *us*
MMS (Maximum Segment Size) of 1024 bytes
TCP algorithm = Reno
Window size = 14 KB

Table 4.2: Characteristics of the modeled network

Listing 4.1 contains the file with the definition of the network topology corresponding to the simulated environment shown in figure 4.1 (level 1). In this environment is defined

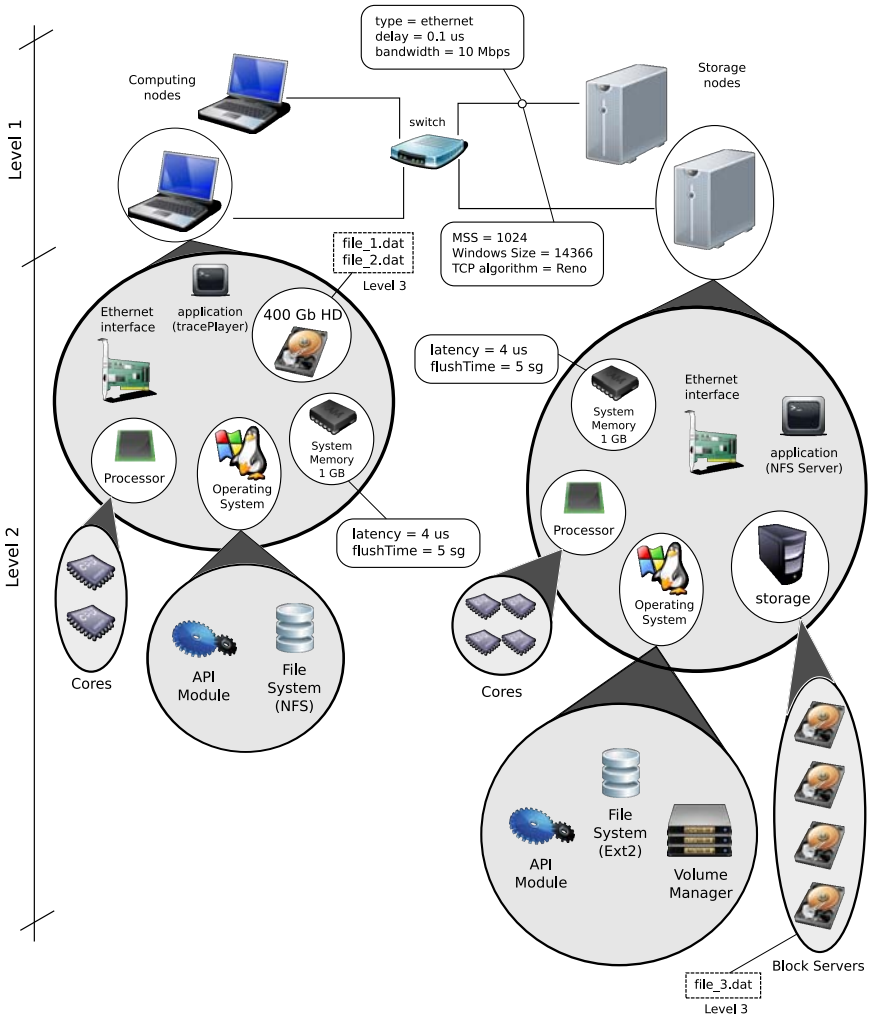


Figure 4.1: Simulated environment

a network that consists of *numClients* computing nodes and *numServers* storage nodes connected to a switch. To interconnect those nodes an Ethernet Gigabit network is used (lines 1-4). The network is configured with a delay of 125 *us* and a bandwidth of 1 Gbps. The number of nodes is configurable by setting the parameters *numClients* and *numServers* (lines 9-10). The configurator module (line 14) sets up the IP addresses to all modules. In line 20 the switch module used in current network is defined. Line 25 and 27 shows the vectors that contain computing nodes and storage nodes modules. Finally, all connections are configured in lines 31-39.

Listing 4.1: Example of network topology

```
1 channel ethernetline
2   delay 125 us;
3   datarate 1000*1000000;
4 endchannel
5
6 module Environment_Example
7
8   parameters:
```

```
9       numClients: numeric const,
10      numServers: numeric const;
11
12    submodules:
13
14      configurator: ParallelNetworkConfigurator;
15        parameters:
16          moduleTypes = "EtherSwitch2 SIMCAN_Node",
17          nonIPModuleTypes = "EtherSwitch2",
18          netmask = "255.0.0.0";
19
20      switch: EtherSwitch2;
21        gatesizes:
22          in[numServers+1],
23          out[numServers+1];
24
25      client: CompleteNode_TCP [numClients];
26
27      server: CompleteNode_TCP [numServers];
28
29    connections nocheck:
30
31      for i=0..numClients-1 do
32        client[i].ethOut++ --> ethernetline --> switch.in++;
33        client[i].ethIn++ <-- ethernetline <-- switch.out++;
34      endfor;
35
36      for i=0..numServers-1 do
37        switch.out++ --> ethernetline --> server[i].ethIn++;
38        switch.in++ <-- ethernetline <-- server[i].ethOut++;
39      endfor;
40
41  endmodule
```

Listing 4.2 shows the parameters file corresponding to the modeled environment (level 2). This file is in charge of customizing the network showed in figure 4.1. For practicality reasons, only the most relevant parts of this file are shown. This file contains a list of pair (parameter, value) for setting up all modules that make up the simulated environment shown in figure 4.1. For instance, this file configures a network that consists of 2 computing nodes (line 2), 2 storage nodes (line 3) and one switch. Each computing node contains 2 CPUs and one hard disk drive (lines 6-7). Disk model is set in line 14. Operating system in client nodes are parameterized in lines 17-30. Moreover, computing nodes use a main memory of 1 GB which is configured in lines 26-30.

Storage nodes contain four CPU cores (line 36) and a RAID system with 5 disks (line 37). The block size used in the RAID system is 4KB (line 628). Each storage node contains a NFS server application listening on port 2049 (lines 40-41). The disk type used is a diskSim model (line 44). Those nodes contain an Ext2 File system, which is configured in lines 54-59.

Finally, several parameters for configuring the TCP settings are located in lines 68-70.

Listing 4.2: Example of parameter file

```
1  [Parameters]
2  environment_Example.numServers = 2;
3  environment_Example.numClients = 2;
4
5  # Client configuration
6  tcp_Network_nfs.client[*].numCPUs = 2;
7  tcp_Network_nfs.client[*].numBlockServers = 1;
8
```

```
# Applications
  tcp_Network_nfs.client[*].appModule[0].appType = "TracePlayer";
  tcp_Network_nfs.client[*].appModule[0].app.traceFile = "trace.txt";

# Disk
  tcp_Network_nfs.client[*].bsModule[*].diskType = "Disk_LI_400GB";

# Operating_System
  tcp_Network_nfs.client[*].osModule.networkServiceType = "NetworkService";
  tcp_Network_nfs.client[*].osModule.cpuSchedulerType = "CPU_Scheduler";
  tcp_Network_nfs.client[*].osModule.memoryType = "MainMemory";
  tcp_Network_nfs.client[*].osModule.ioRedirectorType = "IORedirector";
  tcp_Network_nfs.client[*].osModule.fsModuleType = "FSModule";
  tcp_Network_nfs.client[*].osModule.vmModuleType = "VolumeManagerModule";

  # Memory
    tcp_Network_nfs.client[*].osModule.memory.latencyTime_s = 0.000004;
    tcp_Network_nfs.client[*].osModule.memory.flushTime_s = 30;
    tcp_Network_nfs.client[*].osModule.memory.size_KB = 1048576;
    tcp_Network_nfs.client[*].osModule.memory.blockSize_KB = 8;
    tcp_Network_nfs.client[*].osModule.memory.readAheadBlocks = 2;

  . . .

# Servers configuration

tcp_Network_nfs.server[*].numCPUs = 4;
tcp_Network_nfs.server[*].numBlockServers = 5;

  # Applications
    tcp_Network_nfs.server[*].appModule[0].appType = "NFS_Server";
    tcp_Network_nfs.server[*].appModule[0].app.localPort = 2049;

    # Block_Servers
    tcp_Network_nfs.server[*].bsModule[*].diskType = "DiskSim_Disk";

    # Memory
        tcp_Network_nfs.server[*].osModule.memory.latencyTime_s = 0.000004;
        tcp_Network_nfs.server[*].osModule.memory.flushTime_s = 30;
        tcp_Network_nfs.server[*].osModule.memory.size_KB = 8421376;
        tcp_Network_nfs.server[*].osModule.memory.blockSize_KB = 8;
        tcp_Network_nfs.server[*].osModule.memory.readAheadBlocks = 2;

    # File_System [0]
    tcp_Network_nfs.server[*].osModule.fsModule[0].fsType = "DI_FileSystem";
    tcp_Network_nfs.server[*].osModule.fsModule[0].fs.maxBlocks = 838860800;
    tcp_Network_nfs.server[*].osModule.fsModule[0].fs.diskRatio = 0;
    tcp_Network_nfs.server[*].osModule.fsModule[0].fs.fsType = "ext2";
    tcp_Network_nfs.server[*].osModule.fsModule[0].fs.fsBlockSize_b = 4096;
    tcp_Network_nfs.server[*].osModule.fsModule[0].fs.preLoadFiles = true;

    # Volume_Manager
    tcp_Network_nfs.server[*].osModule.vmModule.blockManager.strideSize_b = 4096;

  . . .

#_TCP_settings

**.tcp.mss = 1024
**.tcp.advertisedWindow = 14336
**.tcp.tcpAlgorithmClass="TCPReno"

  . . .
```

Additionally, in this example have been used files corresponding with the configuration of level 3: server list, file system contents and I/O redirector. Listing 4.3 shows an example

of servers list file. First line indicates the number of server to be loaded. Each next line refers to a server, which contains three fields: IP address, listen port and server ID.

Listing 4.3: Example of server list file

```
3
192.168.0.2:9000:0
192.168.0.3:9000:1
192.168.0.4:9000:2
```

Listing 4.4 shows an example of a file system contents file. First line indicates the number of files to be pre-loaded in the file system. Each next line refers to a file, which contains two fields: a file name and a file size (measured in KB).

Listing 4.4: Example of file system contents

```
3
/local/file_1.dat:1024
/local/file_2.dat:1024000
/remote/file_3.dat:512
```

Listing 4.5 shows an example of an I/O redirector file. First line indicates the number of entries that the I/O redirector module will load. Each next line refers to a partition, which contains three fields. First field is the partition path. Second field indicates whether the current partition is local or remote. Third field indicates the ID, in case of local partition, this ID refers to the file system. Otherwise, this field refers to the server ID, which must be configured in the servers file (see listing 4.3).

Listing 4.5: Example of I/O redirector file

```
2
/local:LOCAL:0
/remote:REMOTE:1
```

4.2 Scaling up modeled environments in SIMCAN

The task of modeling simulated environments used to be a tedious and time-consuming task. Basically it consists on defining the components involved in the simulation and configuring each one for a specific purpose. Thus, the time spent on performing this task is directly correlated with next three factors:

1. Basic knowledge of the language used for defining the environment: Its complexity varies with the simulation platform but for regular users is a hard task.
2. Basic knowledge of the components involved in the simulated environment: Its complexity grows with the number of components included.
3. Error correction: Syntax and semantic errors that would appear during first executions until they are eliminated with a try-and-error process.

In this section we propose several approaches for speed up and alleviate this task as far as possible. Those approaches have not been designed for a specific simulation platform, instead those can be implemented for any simulation platform. Thus, we propose the

next definitions for modeling each component in a distributed system model. Basically, components in a distributed system model can be grouped in three categories:

- Nodes: The main elements. They have different roles like: computing nodes for processing, storage nodes, for managing and store data, etc.
- Communication Devices: Those devices are in charge of interconnecting nodes or other communication devices through communication links.
- Aggregation modules: Organized groups of system components to ease its deploy and management. Examples are node boards (a group of nodes with a switch) and racks (a group of node boards)

Following, in order to describe the process of configuring modeled environments, a formal notation of the components used to model such environments will be described. This notation will be used as the basis for developing a proposed algorithm, which is targeted to organize the steps required for achieving the configuration of the modeled environments. The main objective of this proposed algorithm is to integrate it in a graphic tool aimed to ease the configuration of simulated environments, specifically to novel users that are not used to manage simulation tools. This tool, called SIMCAN Scenario Creator is described in detail in section 4.3.

Another target where this notation can be used, is in the development of a method for automatically partitioning the graph that represents the architecture of any model. The main objective of this method is to parallelize the execution of the model automatically. This is achieved by splitting the whole model in a set of sub-models. Thus, the model can be simulated in parallel by executing the simulation of each sub-model in a computing node. Moreover, this algorithm balance the load of each partitioned sub-model, minimizing as much as possible the number of communications performed between sub-modules in order to reduce the overhead. Both proposed algorithms are described in the section.

Thus, a component in the system can be a node, a communication device or an aggregation module. At the same time, aggregation modules can also contain nodes and communication devices. Basically a model is completely defined by:

- A set of components involved in the model.
- A set of parameters that configures each one of the involved components in the model.
- A graph that defines the connections between components in the model.

A model is defined as:

$$M = \{Z, E\}$$

Such that Z is a finite set of pairs (C, P), where C contains all components of the model M, and P contains a set of parameters that configures components of C. Then, Z can be defined as:

$$Z = \{(C_i, P_i)\} \forall i | (0 < i \leq |Z|)$$

Let C be a finite set of components in the system categorized like nodes N, communication devices D, and aggregation modules A, such that:

$$C = \{N, D, A\}$$

Let N be a finite set of nodes defined as:

$$N = \{(N_i)\}\forall i|(0 < i \leq |N|), N_i \in C$$

Let D be a finite set of communication devices defined as:

$$D = \{(D_i)\}\forall i|(0 < i \leq |D|), D_i \in C$$

Let A be a finite set of aggregation modules defined as:

$$A = \{\{B_i\}, \{R_j\}\}\forall i, j|(0 < i \leq |B|), (0 < j \leq |R|), B_iR_j \in C$$

Let B be a finite set of node boards defined as:

$$B = \{(B_i = (\{N_j\}, D_k)\}\forall i|(0 < i \leq |B|), \forall j|(0 < j \leq |N|), \forall k(0 < k \leq |D|),$$
$$N_j \in N, D_k \in D, \forall B_xB_y \in A|B_x \cap B_y = \emptyset$$

Let R be a finite set of racks defined as:

$$R = \{(B_i)\}\forall i|(0 < i \leq |B|), \forall R_iR_j \in A|(R_i \cap R_j = \emptyset)$$

Let P_i be a finite set of features K such that all $K \in P_i$ characterizes completely the component C_i. Then, P_i can be defined as:

$$P_i = \{(K_j)\}\forall j|(0 \leq j < |P_i|)$$

Let E be a finite set of pairs of components (c_s, c_e) such that for all $c_sc_e \in C$, those pairs defines connections between components in the model. Then, E can be defined as

$$E = \{(c_s, c_e)\}\forall s, e|c_s, c_e \in C, c_s \in D, (c_e \in D \vee c_e \in N)$$

The topology of a model M can be defined as a non-directed graph $G = \{V, E\}$ such that, V is a finite set of pairs that represents the components of the model that are interconnected, defined as:

$$V = \{(Ci)/\forall i|C_i \in C, (C_i \in N \vee C_i \in D)\}$$

The degree of each vertex of G represents the number of communication links associated to a corresponding component in the system, which is denoted by:

$$d_G(v) = |N_G(v)|\forall v \in V$$

where $N_G(v)$ is the number of neighbors of the vertex v, defined as:

$$N_G(v) = \{u \in V / \forall (v,u) \in E\}$$

Using the notation previously described we propose a strategy that will let users building distributed environments easily and quickly. Basically this strategy consists on obtaining the basic information from the user and then, creating the corresponding simulated environments. Algorithm 9 shows the pseudo-code for creating a simulation environment from the initial information provided by the user, which consists basically in the set Z, that contains the set of components that will be simulated in the model, and the graph G that contains the topology of the system.

Algorithm 9 Generating a simulated computer architecture environment

Require: $Z = \{(C,P)\}$ and $G = \{(V,E)/\forall V \in E\}$
Ensure: Output file with the generated model

```
    // Configure each element in the model
 1: for i = 0 to i < |C| do
 2:   if C_i ∈ N then
 3:     writeConfigurationForNode (C_i, P_i)
 4:   else if C_i ∈ D then
 5:     writeConfigurationForCommunicationDevice (C_i, P_i)
 6:   else if C_i ∈ A then
 7:     if C_i ∈ B then
 8:       configureComponentsInNodeBoard (C_i, P_i)
 9:     else if C_i ∈ R then
10:       for all B_j such that (B_j ∈ C_i) do
11:         configureComponentsInNodeBoard (B_j, P_i)
12:       end for
13:     end if
14:   end if
15: end for
    // Establish connections between the corresponding elements
16: for all (u, v) such that (u, v) ∈ E do
17:   writeConnection (u, v)
18: end for
```

Algorithm 10 configureComponentsInNodeBoard(B_x, P_x)

Require: $(B_x, P_x)/B_x \in E \wedge P_x \in P \wedge (B_x, P_x) \in Z$
Ensure: Set of configurations for all components in node Board B

```
1: for all (X_i, P_i) such that (X_i ∈ B_x ∧ P_i ∈ P_x) do
2:   if X_i ∈ N then
3:     writeConfigurationForNode (X_i, P_i)
4:   else
5:     writeConfigurationForCommunicationDevice (X_i, P_i)
6:   end if
7: end for
```

This task becomes harder and more tedious when the number of components involved in the simulation is huge. In terms of performance, executing simulations with a huge number of components (hundreds or even thousands) require a lot CPU power and memory. In order to increase the performance of those kinds of scenarios, parallel simulation can be a feasible solution. Basically, parallel simulation consists on splitting the model M in a set of logical partitions, where each partition is in charge of simulating a set of elements instead of the complete model. Formally, let T be a finite set of partitions, where the number of logical partitions used for executing the simulation is given by $numPartitions$, such that:

$$T = \{(X_i)\}\forall i|(0 < i \leq numPartitions)$$

Where X_i represent a partition, defined as:

$$X_i = \{\{C_i\} \cap \{E_j\}\}\forall i|(0 < i \leq |C|), \forall j(0 < j \leq |E|),$$
$$\forall k|C_iC_k \in C, (C_i \in X_i, C_k \in X_k, C_i \cap C_k = \emptyset)$$

Then, once the model has been completely distributed among partitions, each partition will be executed in a corresponding host. Thus, the overall simulation performance will be increased, because a set of hosts are working in parallel and sharing resources such as CPU and memory for executing the simulation. However, the speedup of parallel simulation is not linear because the overhead caused for communicating and synchronizing the components located in different partitions produces a drop of performance. Formally, let $time_{local}$ (see equation 4.1) be the time needed for sending a message between two components located in the same partition, and let $time_{remote}$ (see equation 4.2) be the time needed for sending the same message between two components located in different partitions, defined as:

$$time_{local}(c_A, c_B) = time(c_A, c_B)|c_A \in X_i, c_B \in X_i \tag{4.1}$$

$$time_{remote}(c_A, c_B) = time(c_A, c_B)|c_A \in X_i, c_B \in X_j, c_A \notin X_j, c_B \notin X_i \tag{4.2}$$

Then $time_{local} << time_{remote}$ because hosts use a communication network for interchanging messages between them, and this way synchronizing and communicating components located in different partitions. This process is much costly than sending a message between components located in the same host, which usually consists on invoking a method. This is mainly the cause because the performance of a parallel model is not increased linearly. Then, a good partitioned model reduces the number of communications between components located in different partitions as much as possible, which is a hard and complex task.

However, configuring parallel simulations hamper the task of generating simulated environments, because the user is who has to split and configure by hand each partition $X_i \in T$. Moreover, another factor that adds complexity to this task is the fact of balancing the distribution of the model among a set of partitions. This balancing consists on adjust the ratio between the number of components simulated per partition and the number of communications between partitions.

In order to ease this task, we propose an approach for distributing a model among a set of partitions, keeping balanced the number of components per partition and the number

of communications between different partitions. With the purpose to accomplish this task, the user has to provide the graph that contains the topology of the model G, and the number of partitions used for splitting the model. The result will be a split model ready to be executed in parallel. Due to the number of communications is directly correlated with the applications executed in the nodes, it is not possible to maintain a perfect balance until an exhaustive study of all applications executed in the model is performed. Then, due to the objective of this work is to ease the task of creating distributed environment models to be executed in parallel, our approach supposes a uniform number of communications between nodes. Algorithm 11 shows the pseudo-code corresponding to this approach.

Figures 4.2 and 4.3 shows an example of how a model is distributed among 4 partitions ($T = \{X_1, X_2, X_3, X_4\}$), balancing the ratio between the number of components to be simulated in each partition and the number of communications between different partitions.

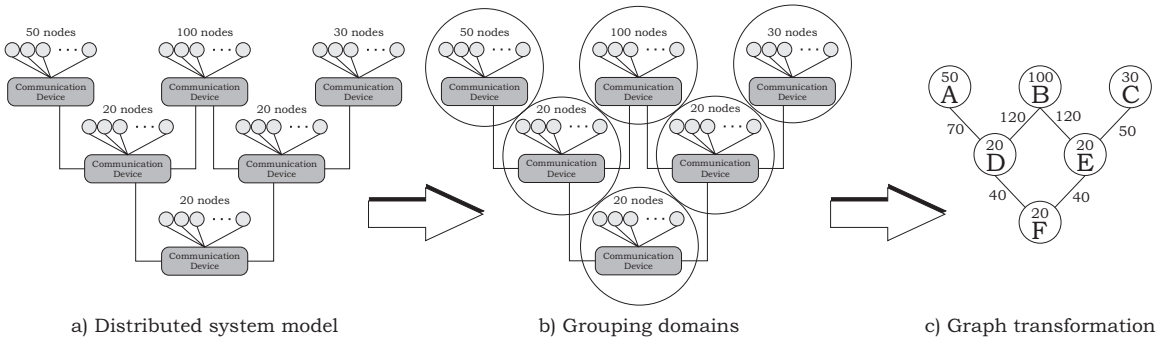


Figure 4.2: Model transformation process

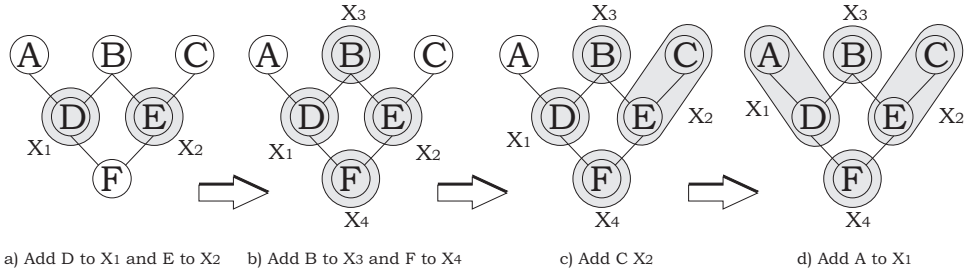


Figure 4.3: Steps for distributing model of figure 4.2.a among 4 partitions

First, the user must provide a model (see figure 4.2.a). Then, this model has to be divided in domains, where each domain contains a communication device and a set of nodes directly attached to this communication device (see figure 4.2.b). The purpose of those domains is to assure that each section of the model where the probability of a high number of communications can be produced, be executed in the same partition in order to reduce overhead. Once the model has been divided into domains, then it is transformed to a weighted graph, where each vertex in the graph represents a domain, and each edge represents a communication link between two domains (see figure 4.2.c). Each vertex has associated a number that represents the number of nodes located in that domain. Moreover, the weight of each edge represents the communications than can be produced directly between the attached domains.

The generated graph (see figure 4.2.c) can be defined as an edge weighted graph, that is, G^{α} is a graph $G = \{V, E\}$, together with a weight function $\alpha : E_G \to \Re$ on its edges, such that V is a set of vertex and E is a set of edges. In this example $V = \{A, B, C, D, E, F\}$ and $E = \{AD, BD, BE, CE, DF, EF\}$.

Following, for a better understanding of algorithm 11 some functions used in it are described. Let γ be a function that calculates the number of communications between 2 nodes, such that:

$$\gamma(v, G) = \sum_{i=0}^{d_G(v)} \alpha(vu) \forall u \in N_G(v);\ \gamma(X_i, G) = \sum \alpha(vu) \forall (v, u) \in E / (i \neq j \wedge v \in X_i \wedge u \in X_j)$$

Let $\delta(v)$ be a function that calculates the number of nodes of a given vertex v, such that:

$$\delta(X_i, G) = \sum \delta(v, G) \forall v \in X_i$$

Let $\rho(v)$ be a function that calculates the ratio between the number of nodes and the number of communications of a corresponding vertex v, such that:

$$\rho(v, G) = \frac{\delta(v, G)}{\gamma(v, G)}$$

Let $nodes_G$ be the total number of nodes in the graph G, such that:

$$nodes_G = \sum \delta(v) \forall v \in V$$

Let $nodesAvg_G$ be the average number of nodes per partition in the whole model, defined as:

$$nodesAvg_G = \frac{nodes_G}{|T|}$$

Let $comm_G$ be the total number of communications in the graph G, such that:

$$comm_G = \sum \gamma(e) \forall e \in E$$

Let $commAvg_G$ be the average number of communiaction in the whole model, defined as:

$$commAvg_G = \frac{comm_G}{|T|}$$

Let β be a function that calculates how balanced is the model, such that:

$$\beta(X_i, G) = |nodesAvg_G - \delta(X_i, G)| + |commAvg_G - \gamma(X_i, G)|$$

Then, the balance of T is given by:

$$\beta(T,G) = \sum \beta(X_i,G) \forall X_i \in T$$

Function β returns a value that indicates how well partitioned is the model among the set of partitions T. The closer this value to 0, the better balanced is the model. Then $\beta(T) = 0$ means that the model is perfectly balanced among partitions of T. Basically the idea is to keep balanced in each partition the ratio between the number of components to be simulated and the number of communications with different partitions. Table 4.3 shows all steps performed by algorithm 11 to distribute the model of figure 4.2.a. Also, figure 4.3 shows where each one of those domains is placed in a corresponding partition until the complete model is distributed.

Step	X_1	X_2	X_3	X_4
add (D, X_1)	δ=20,γ=30,β=160	δ=0,γ=0,β=170	δ=0,γ=0,β=170	δ=0,γ=0,β=170
add (E, X_2)	δ=20,γ=30,β=160	δ=20,γ=210,β=140	δ=0,γ=0,β=170	δ=0,γ=0,β=170
add (B, X_3)	δ=20,γ=30,β=160	δ=20,γ=210,β=140	δ=100,γ=240,β=170	δ=0,γ=0,β=170
add (F, X_4)	δ=20,γ=30,β=160	δ=20,γ=210,β=140	δ=100,γ=240,β=170	δ=20,γ=80,β=70
add (C, X_2)	δ=20,γ=30,β=160	δ=50,γ=160,β=60	δ=100,γ=240,β=170	δ=20,γ=80,β=70
add (A, X_1)	δ=70,γ=160,β=60	δ=50,γ=160,β=60	δ=100,γ=240,β=170	δ=20,γ=80,β=70

Table 4.3: Steps performed for partitioning the model

Initially, a set of vertex with the maximum degree is calculated from G. In this case, both vertex D and E have a degree of 3. Due to $\rho(D) < \rho(E)$, vertex D is the candidate to be associated to a corresponding partition, which in this case is X_1. Then, vertex E is associated to partition X_2 (see figure 4.3.a).

Due to there is no more vertex with degree 3, then a set of vertex with degree 2 are calculated (B and F). Due to $\rho(B) < \rho(F)$, vertex B is the candidate to be assigned to a corresponding partition. B is assigned to partition X_3 and F to partition X_4 (see figure 4.3.b).

Next step consists on calculating vertex with degree 1 (C and A). In order to maintain balance, vertex C is assigned to partition X_2 (see figure 4.3.c) and vertex A is assigned to partition X_1 (see figure 4.3.d). Finally, the model has been distributed in 4 partitions, keeping balanced the number of components per partition and the number of communications between partitions. The ratio between number of components and number of remote communications is 0.43 for X_1, 0.31 for X_2, 0.41 for X_3 and 0.25 for X_4, which indicates a good partitioning.

4.3 The SIMCAN Scenario Creator tool

The strategies described in section 4.2 have been implemented in a tool called SIMCAN Scenario Creator. This tool is a GUI-based Java application targeted to manage and generate models for the SIMCAN simulation platform. The main goal of this tool is two-fold. First, hiding all low-level details, including the language used for implementing components and creating simulated environments. Second, ease the generation and customization of distributed system models. Following, the main features of this tool are described.

- Automatic browsing and loading of the simulator's core modules.

- Generation and management of main building blocks.
- Managing and executing simulations.

4.3.1 Automatic browsing and loading of the simulator's core modules

SIMCAN Scenario Creator provides an intuitive graphical interface for managing simulation models. One of the main assets provided by this tool is that users can load modules and browsing among them easily by using a graphical interface.

The core engine of this simulation platform is composed by a set of modules, as was previously described in section 3.2. Those modules are hierarchically nested, where a module can contain zero or a set of modules. For instance, a node in SIMCAN (see figure 3.3) contains other modules such as an operating system, a set of applications, a set of processors, a set of block servers, etc. Each one of those modules contains a list of parameters, which must be configured for obtaining a concrete behavior of each component. Due to number of modules existent in the current version of SIMCAN is high, configuring a distributed system model by hand becomes a hard task, especially for novel users.

For instance, a large system model can contain hundreds of parameters. Traditionally, users had to write a set of text files (see section 4.1) that contain those parameters and the definition of each involved component in the model. Moreover, the only way to know the list of parameters of each module is to read the documentation provided by SIMCAN.

With the purpose of ease this task, and saving time and effort, this application shows a complete list of modules and parameters automatically. For example, figure 4.4 shows the modules inside a node (left panel), and the list of modules inside the operating system module (right panel). Thence, users can see a complete list of components to be modeled. Once users select a concrete component, this application shows the list of parameters required for configuring the selected component.

In order to provide the last updated version of this repository, this tool performs scans into the SIMCAN location to refresh the list of components and providing the last version of each one. This feature is very useful because the SIMCAN's core engine is updated by adding, removing or even modifying a module, those changes are automatically reflected in this tool without modifying the source code of this application. This is achieved by scanning the SIMCAN's core engine and generating a set of XML files. Those files contain updated information about the modules and their corresponding list of parameters of the current version of SIMCAN. Thus, those modules with their corresponding parameters are shown in the GUI automatically, and ready to be configured by users to create simulation models.

4.3.2 Generation and management of main building blocks

The SIMCAN simulation platform is targeted to model and simulate HPC systems. Thus, the main building blocks of those systems are: nodes, switches and aggregation components, like racks (see section 3.2).

Therefore, each system model may consist of different instances of those building blocks, where an instance is a pre-configured building block module with a set of specific parameter values. In the case of nodes, HPC system models can contain computing nodes

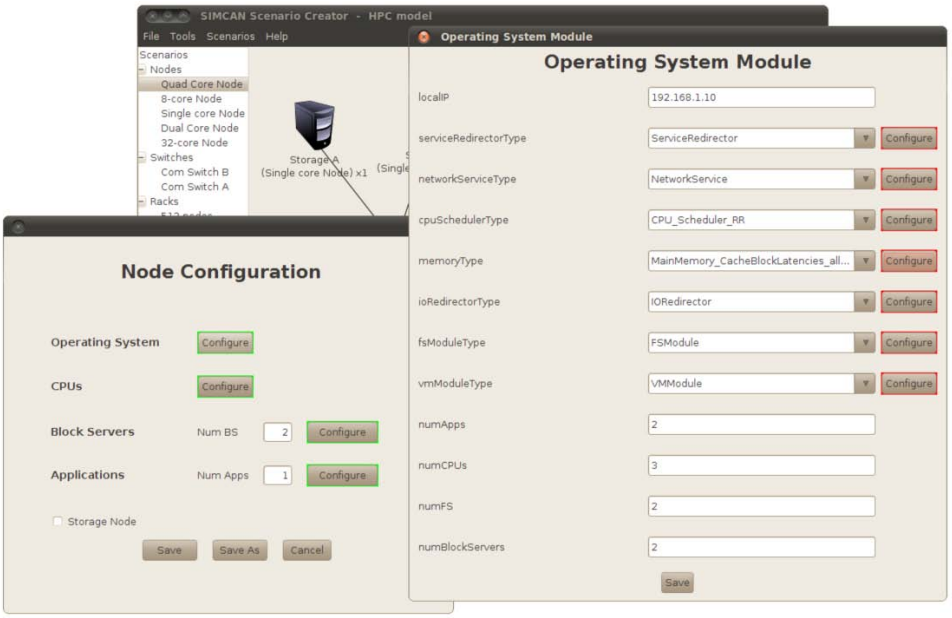


Figure 4.4: Configuration of a node in SIMCAN

and storage nodes, where each type of node can be configured with a set of specific features, like the number of CPU cores, the speed of CPU cores, the amount of memory, the configuration of file systems, the number of disks, etc. In the case of racks, each type of rack can be configured for hosting a different number of node boards, and a different number of nodes. Finally, in the case of switches, each type of switch can be configured by using a different transmission rate, or a different processing time for each processed packet.

Traditionally, each instance of those building blocks is configured by using a set of parameter values stored in a configuration file. Thence, when users need to use the same instance in different environments, a new configuration file must be rewritten by hand using the same parameters. This task can be achieved for environments that use a reduced number of instances, but those users that require a wide range of instances for covering several architectural configurations have to spend a considerable amount of time to write by hand the configuration files.

This tool let users manage a repository of the main building blocks to model HPC systems. This repository contains a set of instances, which can be used for users to build the required HPC model (see left frame of figure 4.5). Thence, users do not have to rewrite the same parameters in a configuration file each time they have to use the same instance. Moreover, those instances are saved to disk in xml files. Thereof, each time a user load a concrete instance and use it to build a model, the list parameters of the loaded instance is written to the text configuration file that contains the simulation. Thus, if an instance is modified, each time the simulation is launched, a new configuration file containing those changes will be created, and the simulation will use the latest version of the instance.

Those instances can be used both for building a simulation model and to generate new instances by modifying some parameters. The most typical case is to use an instance of a node and to modify the set of applications to be executed in this node (see figure 4.6).

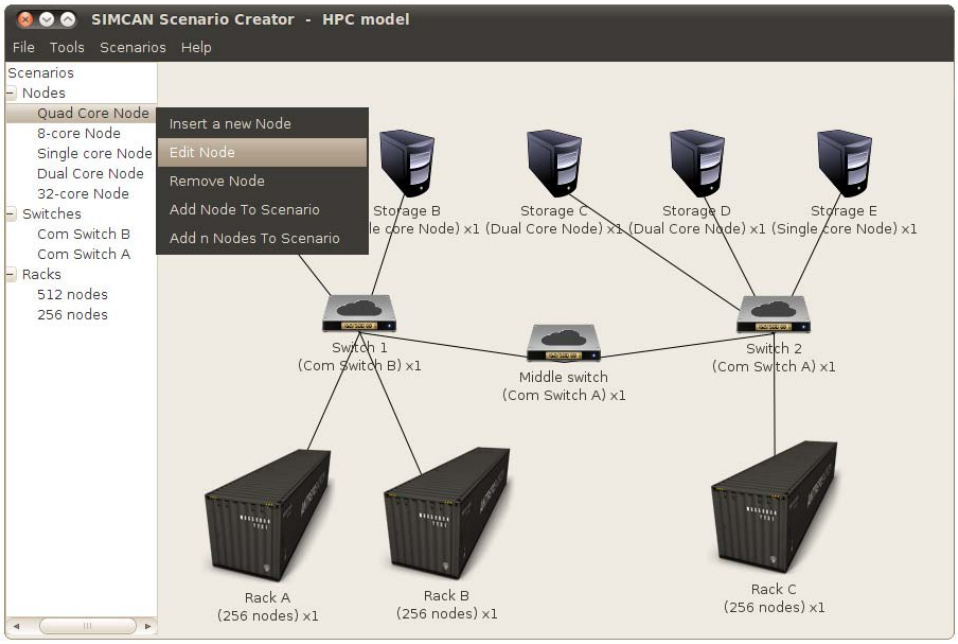


Figure 4.5: Managing a repository of main building blocks

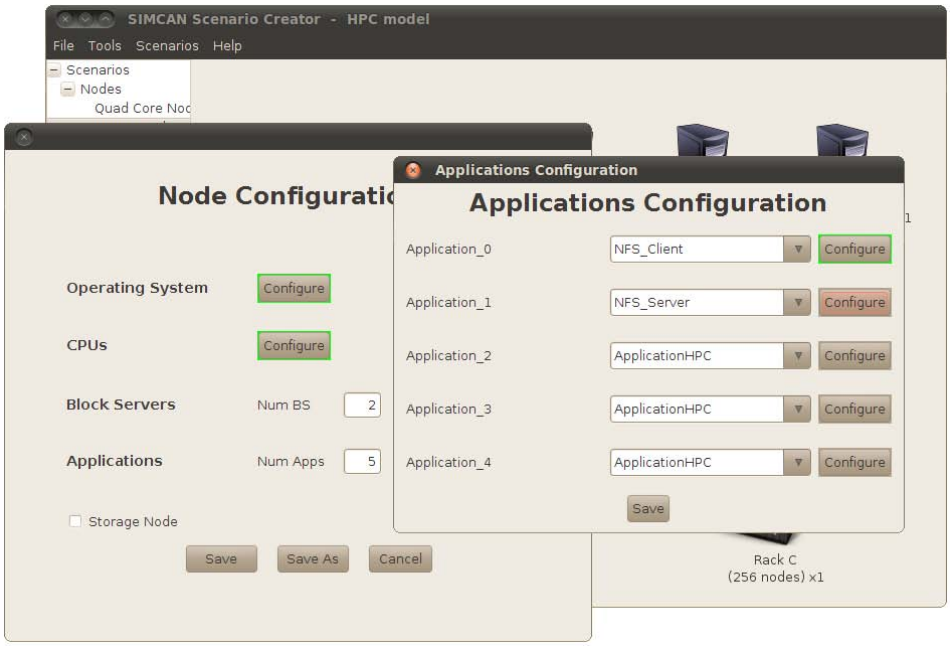


Figure 4.6: Configuring a set of applications simulated in a node

4.3.3 Managing and executing simulations

Finally, another important feature of this application is the possibility of designing graphically the topology of the system to be modeled. This is basically how the components involved in the model are connected.

Traditionally, this topology is defined in text files using module pairs to represent a connection, which hampers the visualization of the model. Basically, each pair of modules represents that those modules are connected in the system. Moreover, the definition of this connection can be parameterized using the corresponding list of parameters such as bandwidth, transmission delay and error rate (see figure 4.7).

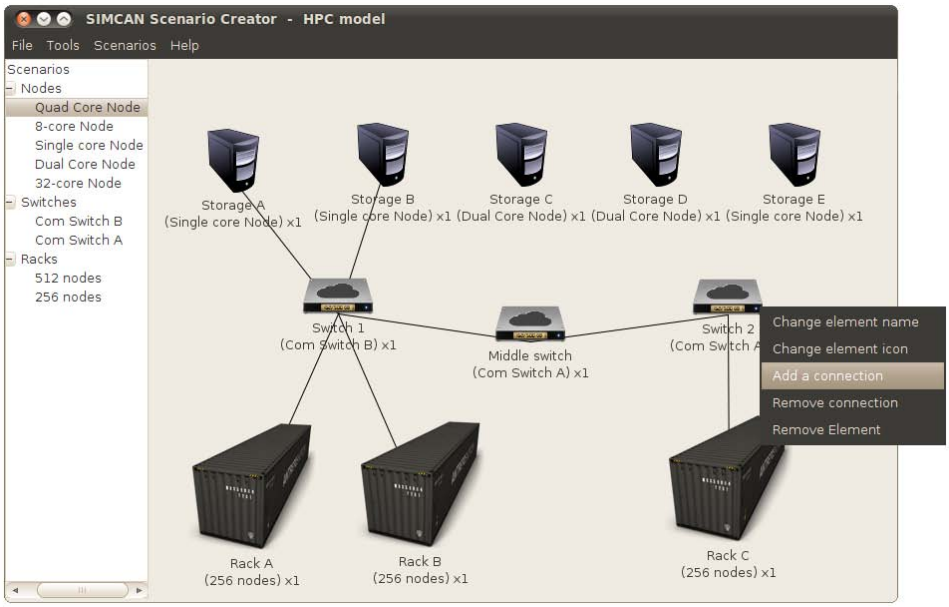


Figure 4.7: Configuring connections between modules

Thence, users can visualize easily the architecture of the model and perform changes quickly just using the drag-and-drop actions in the GUI, which is much easier and faster than performing those changes in text files by hand.

Furthermore, this application performs a validation of the model created by users. Once the model is finished and before creating the configuration files that represent the model, the validation process is achieved in three different phases. First, the application checks that each parameter has been set, due to empty parameters without value generates an error when the simulation is launched. Second, the application checks the type of each parameter, due to the type of a parameter can be boolean, integer or string. Although the GUI indicates users the type of each parameter, it is responsibility of the user to type the right parameter in each text box. Figures 4.4 and 4.6 show those modules that have not been configured yet with red background, and those modules that have been configured in green background. Finally, the application checks that all modules are connected correctly.

A collection of distributed environments can be managed using this tool, which let users change previously configured scenarios, load and save new created models. For large distributed models, this tool also provide the possibility to automatically distributing those models for performing parallel simulations, which saves great amounts of time for users. Thus, the model is distributed among a set of partitions using algorithm 11 described in section 4.2. All IPs for nodes involved in the model are generated automatically.

In order to analyze the usefulness of the SIMCAN Scenario Creator, 4 users have built a distributed model using both the traditional method and this presented tool. Traditional

Algorithm 11 Automatic partitioned of the model for parallel simulations

Require: numPartitions, $G = \{(V, E)\}$ and $numPartitions$
Ensure: T
// Initialization
1: maxDegreeSet $\leftarrow \emptyset$
2: currentVertex $\leftarrow \emptyset$
// Transform the model into a weighted graph
3: $G_D \leftarrow$ divideModelInDomains (G)
4: $G^{\alpha} \leftarrow$ generateWeightedGraph (G_D)
5: $V_{aux} \leftarrow$ V
// Assign each domain $\in V$ to a partition $\in T$
6: **while** $V_{aux} \neq \emptyset$ **do**
7: maxDegreeSet $\leftarrow \forall v \in V / d_G(v) == max(d_G(v))$
// Calculate the set of vertex with higher degree and lower ρ
8: currentVertex $\leftarrow \forall v \in maxDegreeSet / \rho(v) == min(\rho(v))$
// Calculate the partition that provides better balance for adding $currentVertex$ in
9: **for all** X_i such that $(X_i \in T \wedge (0 < i \leq numPartitions))$ **do**
10: $k \leftarrow min(\beta(add(X_i, currentVertex), G^{\alpha}))$
11: $X_k \leftarrow add(X_k, v)$
12: **end for**
// Update sets
13: currentVertex $\leftarrow \emptyset$
14: maxDegreeSet $\leftarrow$ remove (maxDegreeSet, currentVertex)
15: $V_{aux} \leftarrow$ remove (V_{aux}, currentVertex)
16: **end while**

method basically consists on writing the topology of the model using text files. In this case, two different text files are needed, one for defining the topology of the model, and other file for customize the parameters of each component involved in the model. The environment to be modeled consists of 4096 nodes and 40 switches. Those nodes contain both storage and computing systems. Moreover, users have to manually distribute the model in 2, 4, 8 and 16 partitions for performing parallel simulations.

Advance user is a PhD student of Computer Science. This user has a high degree of expertise in developing components for the SIMCAN simulation platform. Intermediate user 1 is a Dr. in Computer Science who uses occasionally SIMCAN but they are not used to develop components for SIMCAN. Intermediate user 2 is a PhD student of Computer Science who uses occasionally this simulation platform for performing her research studies. Finally, novel user is a graduate student in Computer Science. This user has one year of expertise working with simulators, but he did not know SIMCAN at all. Table 4.4 shows the amount of time spent for each user to create the described distributed model.

User	Create a model by hand	Create a model using SIMCAN Scenario Creator
Advanced user	0.45	0.15
Intermediate user 1	4	0.25
Intermediate user 2	3.5	0.2
Novel user	9	0.35

Table 4.4: Amount of time (in hours) needed for creating a large parallel distributed model

4.4 Summary

In this chapter has been described the process for modeling environments using the SIMCAN simulation platform.

Moreover, several approaches for creating large distributed models in simulation platforms have been presented. Those strategies also include automatic partitioning of simulated models for performing parallel simulations.

Those strategies have been implemented in a tool called SIMCAN Scenario Creator, which is targeted to create environments for the SIMCAN simulation framework. Moreover, this tool demonstrates that users can obtain a good performance by using this tool for generating large distributed models. Also, those scenarios can be created in less time than using traditional methods. In some cases, amount of time saved is 25 times less.

Chapter 5

Strategies and SIMCAN facilities for modeling applications

In order to predict the performance of any system, apart from modeling and simulating the architecture of such system, the applications to be executed in those systems must be also modeled and simulated. However, there are several issues that make the modeling of applications a hard and complex task, like data-dependent computation times, inter-process communications and synchronization delays, and other architecture-specific timing information.

In this chapter, three different strategies for modeling and implementing applications in the SIMCAN simulation platform are presented together with some facilities to implement them. Using those strategies, the simulation of a wide range of applications executed in the required system can be achieved.

5.1 Introduction

Currently, designers and users of distributed systems have the difficult task of exploiting the resources that are available in those kinds of environments. The good use of them depends highly both of the design of the system architecture and the design of the application to be executed. A wrong system design will not let users exploit totally the available resources. In the other side, a wrong algorithm won't use the resources appropriately. In most cases, distributed applications have different behaviors on very large scales. It is required to match the structure of the applications with the structure of the system architecture where applications will be executed. Evaluating, analyzing and predicting the performance of those applications in distributed systems is a difficult challenge, due to the complex interaction between the application characteristics and architectural features.

In order to predict the performance of any system, modeling and simulating the system architecture is not enough. The application to be executed in the system to be simulated must be also modeled. However, there are several issues that hamper modeling applications, like data-dependent computation times, inter-process communications and synchronization delays, and other architecture-specific timing information.

Even when a good model of a given application has been designed and implemented,

it must achieve some requirements such as the computational resources needed to be simulated, and the level of accuracy. The model of a given application that requires high execution times (for example hundreds or thousand times more than the real execution) is useless. In some cases, simulation provides high detailed and accurate information of the system performance, but this alternative can be computationally expensive, often forcing the simulator to use a less detailed simulation model and achieving a loss of accuracy, with the purpose of executing the simulation much faster. However, a model that provides a poor accuracy in the obtained results is useless too.

The key to obtain a good application model is to balance the level of abstraction of the application's characteristics and the level of computation required to execute the simulation. Real Applications can be modeled with the level of detail required. Thus, such models go from very basic schemas to complete ports. Therefore, the simpler the model is, the more difficult is to assure a good characterization and the faster the execution of simulation. By the contrary, the more detailed the model is, the easier is to assure a good characterization and the slower the execution of simulation.

There is a lot of state-of-the-art about the modeling and the characterization of applications. In this chapter we propose a set of strategies for modeling applications in SIMCAN. Furthermore, we present all the facilities that SIMCAN provides to ease the modeling of new applications. Those facilities are focused on two main groups: facilities to ease the coding of new application models, and facilities to ease the use of the architecture resources (CPU, memory, storage, and communications).

5.2 Techniques for modeling applications in SIMCAN

In order to model and simulate a given application in the SIMCAN simulation platform, two basic tasks must be achieved. First, the behavior of the given application has to be modeled. Second, that model has to be translated to the simulation environment.

This section describes three techniques in order to fulfill those tasks: trace-driven simulation, modeling a generic graph that represents the behavior of the application and coding the application from scratch in the SIMCAN simulation platform. Moreover, the corresponding advantages and disadvantages of each technique are described in detail.

5.2.1 Trace-Driven techniques

Trace-Driven simulation techniques are widely used in the simulation field for many reasons. One of the most important reasons is that this technique can be used in a simulated environment easily in a relatively short period of time. Another important characteristic of this technique is the possibility of reproducing quickly real workloads in simulated environments.

Basically, two elements are needed in order to use this technique in SIMCAN: the trace that represents the behavior of the application to be modeled, and the module that parses the trace and translates it to the simulated environment.

The trace file must contain the relevant information needed for reproducing the application's behavior. In most cases, this trace contains a sorted list of operations with the most relevant parameters. The more detailed trace, the more accurate results will be

obtained in the simulation.

The process for obtaining the trace will depend highly of the nature of the application to be simulated. A sequential application is relatively easy to trace, but this process becomes more difficult and complex when the application to trace is a parallel application, where several processes which are executed in different nodes, are continuously interchanging messages across a communication network.

Handling and creating traces that represent the behavior of applications is currently a fashion topic of high level of interest in the research community. Thus, in literature there are a lot of methods for managing those kinds of traces. In this work have been developed two different techniques for working with traces, depending of the nature of the calls performed by the application to be traced. First method is valid for capturing the operating system calls. Second method is in charge of tracing the calls performed by an external library, like MPI.

In order to trace the operating system calls, the *strace* command is used. Using this command, the application is executed in a Real Environment and all the system calls executed are handled, including its corresponding parameters. Then, those calls are stored in a trace file. The general syntax for this kind of trace is shown in figure 5.1.

```
timeStamp  operation  (param1, param2, ..., paramN) = result <exec. Time>
```

Figure 5.1: Trace syntax of sequential applications

Each call in the trace contains the following information:

- *timestamp* is the concrete instant when the system call is invoked by the application.
- *operation* is the name of the system call invoked by the application.
- *parameter list* is a list that contains the parameters which the call has been invoked.
- *result* is the result obtained in the execution of the call. In most cases, this result is an integer that shows whether the execution has been performed successfully, or by the contrary, some error happened during its execution.
- *exec. Time* is the amount of time spent of executing the corresponding call.

Tracing the calls performed by an external library follows a different schema because those calls are not captured by *strace*. In this case a wrapper linked to the library is needed, which contains the set of calls to be traced. For example, in order to trace MPI calls, the execution trace is captured by a wrapper program that uses MPI profiling libraries, like the MPE library [GL98]. MPE is a set of profiling libraries to collect information about the behavior of MPI programs. The complete process is illustrated in figure 5.2.

First (1), both the MPI application and the corresponding wrapper are compiled and linked. The code of the MPI application has not been modified at all. Next, the MPI application has to be launched (2). During the execution of the application, the wrapper handles all MPI calls, including the corresponding timestamps at the beginning and the end of each call. Thus, using those timestamps this trace is written to a file. Finally (3), the

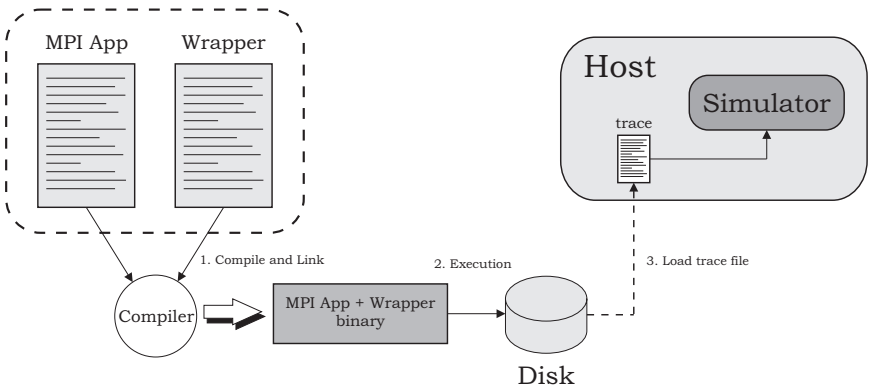


Figure 5.2: Trace handling of parallel applications

execution trace file is read by the simulator, which reproduces exactly the same operations on the simulated environment.

The SIMCAN simulation platform supports a subset of the MPI communication routines, such as point-to-point and collective communications. The general syntax for this trace is shown in figure 5.3.

```
timeStamp  [sourceProcess] MPIcall destProcess  param1, param2,..., paramN
```

Figure 5.3: Trace syntax of parallel applications

Each call in the trace contains the following information:

- *timeStamp* is the concrete instant when *sourceProcess* executes *MPI_Call*.
- *sourceProcess* is the rank of the process that executes *MPI_Call*.
- *MPI_Call* is the MPI call to be executed. When this call contains the suffix INIT, it means that *sourceProcess* starts the execution of such call (i.e. MPI_RECV_INIT). When the MPI call contains the suffix END, it means that such call has been successfully performed (i.e. MPI_RECV_END).
- *destProcess* is the rank of the destination process. This information is only written in those MPI calls that require this information, like MPI_SEND. Other MPI calls, like MPI_BARRIER or MPI_BCAST do not require this information.
- *parameters* is the list of parameters of the invoked call. For example, MPI_Send contains the number of bytes to send to *destProcess*.

An execution of a Real Application contains several processes running in parallel. Each one of those processes has associated a rank number that identifies the process. The Simulated Environment contains the same number of simulated processes that the number of processes used by the Real Application. Thus, each one of those simulated processes contains a rank number, like the Real Application. Therefore, simulated processes read

the same execution trace file, but only executes the MPI calls where the *sourceProcess* coincides with the rank of the simulated process.

The second element required in order to use this technique in SIMCAN is a module that parses those traces and translates them to the simulated environment. Basically this module performs two tasks. First, this module reads and translates the current call to be executed. This call has a corresponding matching pair in the simulated environment which will be invoked using the parameters stored in the trace. Then, once this call has been executed in the Simulated Environment, this module has to wait for its response.

Second, this module has to calculate the amount of time existent between the current call and next call. With the purpose of simulating the computing time of each process, the simulator calculates the difference of time between two consecutive MPI calls. Δ indicates the spent time between two calls. The method for calculating this computing time differs whether the execution of the application is sequential or parallel.

For sequential applications, the difference of time between call i and call i+1 is calculated as shows equation 5.1.

$$\Delta_i = (timeStamp_i + execTime_i) - timeStamp_{i+1} \tag{5.1}$$

where $timeStamp_i$ is the concrete instant when call i is invoked, $execTime_i$ is the amount of time needed for completely executing the call i, and $timeStamp_{i+1}$ is the concrete instant when call i+1 is invoked. Otherwise, for parallel applications the difference of time between the call i, and $timeStamp_{i+1}$ is calculated as shows equation 5.2.

$$\Delta_i = time_{Call\_i+1\_INIT} - time_{MPICall\_i\_END} \tag{5.2}$$

where $time_{Call\_i+1\_INIT}$ is the concrete instant when call i+1 is invoked for the application, and $time_{MPICall\_i\_END}$ is concrete instant when call i has finished its execution.

Then, if a process p executes n MPI calls, the processing time of process p is given by the equation 5.3.

$$processingTime_p = \sum_{i=1}^{k=n-1} \Delta_i \tag{5.3}$$

Instead of generating those traces by capturing the behavior of a corresponding application, those traces can be generated synthetically. The main problem of using synthetic traces is that the behavior of the application is not as accurate as generating the trace by capturing the corresponding operations.

In general, this technique provides an easy method for reproducing the behavior of applications in a Simulated Environment. In one hand, this technique has serious practical shortcomings. First, tracing complete benchmark executions is unfeasible because it requires the storage of billions of instructions. Second, simulation time is also prohibitive for such huge traces; especially if traces are used to evaluate various processor configurations for various workloads, which requires many simulation runs. Depending of the level of detail of such traces, the results obtained can be more or less accurate. In the other hand, this method is very useful for validating a model, because the behavior of a Real

Application can be replayed quickly and accurately in a Simulated Environment using its trace.

In conclusion, this method is useful for validating relatively small environments, but it can be impractical for large distributed environments, because tracing the complete system can be more difficult than simulating it using another technique, that provides more accurate results than this one.

5.2.2 Using pre-defined generic application models

In the distributed computing field, there are some applications that have a similar behavior pattern. This similarity is given by:

- The way those applications performs the corresponding computation.
- How the processed data is distributed among a set of processes.
- How the processed data is treated in the storage system.

We propose to implement those applications patterns using generic parameterized graphs, which represent the behavior of a set of applications. The idea is the following: First, we classify the processes of the application upon their behavior. Second, for each process behavior, a parameterized graph is design to model such behavior.

Thus, each user can configure the parameters of the graphs that represent the application pattern, in order to model specific applications that fit into the pattern. Basically, each graph G contains a finite set of vertex (stages) V and a finite set of edges (operations) E such that:

$$G = \{V, E\}$$

where each vertex $v \in V$ represents a stage, and each edge $e \in E$ represents the execution of a concrete operation. The idea of this technique is that each process p executed in the application to be modeled passes through among a set of stages in the graph. The set of stages and operations performed by a process p in the graph G is called path of behavior of p, such that:

$$path_G(p) = \{V_i, E_j\} | \forall i, j (0 < i \leq |V|), (0 < j \leq |E|), V_i \in V, E_j \in E$$

Then, in order to cross $path_G$, process p must performs a set of concrete operations E_j, whereof each time p performs an operation, current stage of p (V_i) will change to the stage associated to such operation (V_{i+1}). The sub-graph ($path_G$) depends directly on the configuration of the graph previously set by the user. The degree of each vertex v of G represents the number of operations that a process can invoke where such process is placed in a corresponding stage, such that:

$$d_G(v) = |N_G(v)| \forall v \in V$$

where $N_G(v)$ is the number of neighbors of the vertex v, defined as:

$$N_G(v) = \{u \in V / \forall (v,u) \in E\}$$

Each graph must fulfill a set of requirements. First, each graph must contain a start stage $E_{start} \in E$, and an end stage $E_{end} \in E$. Thus, E_{start} is the stage where all processes starts when the simulation begins. Otherwise, E_{end} is the stage where each process finishes one iteration. Finally, graph G must be a cycle graph, such that G must contain a connection between E_{start} and E_{end}. Therefore, each process may perform several iterations by crossing a concrete number of times a given path in G.

The operations reflected in the graph are mainly operations that each process can perfectly perform in the simulation platform. Those operations are gouped in four groups (see section 3.4), depending on the concrete system they belong: computing system, memory system, storage system and network system. The application is then simulated by using a concrete configuration of the graph.

Figure 5.4 shows an example of a graph for modeling a specific kind of HPC applications. Some applications that can be modeled using this graph are BIPS3D [FSI+07] and STEM-II [MSM+01]. In this example two kinds of processes are defined: coordinator processes and compute processes. First ones are in charge of synchronizing a group of compute processes. Also, coordinator processes can read, compute and write data. Second ones are mainly in charge of computing the data. Also, this kind of processes can read and write data. Depending on the application to be modeled, each process will perform concrete tasks. The capability for relying tasks to each process makes the proposed method, flexible enough to cover the behavior of many existing HPC applications. The graphs associated to both kinds of processes have been merged into one graph in order to make the example more concise.

This example (see figure 5.4) shows the state of each process during the execution of the simulated application. Each circle shows the state of the coordinator process (top) and

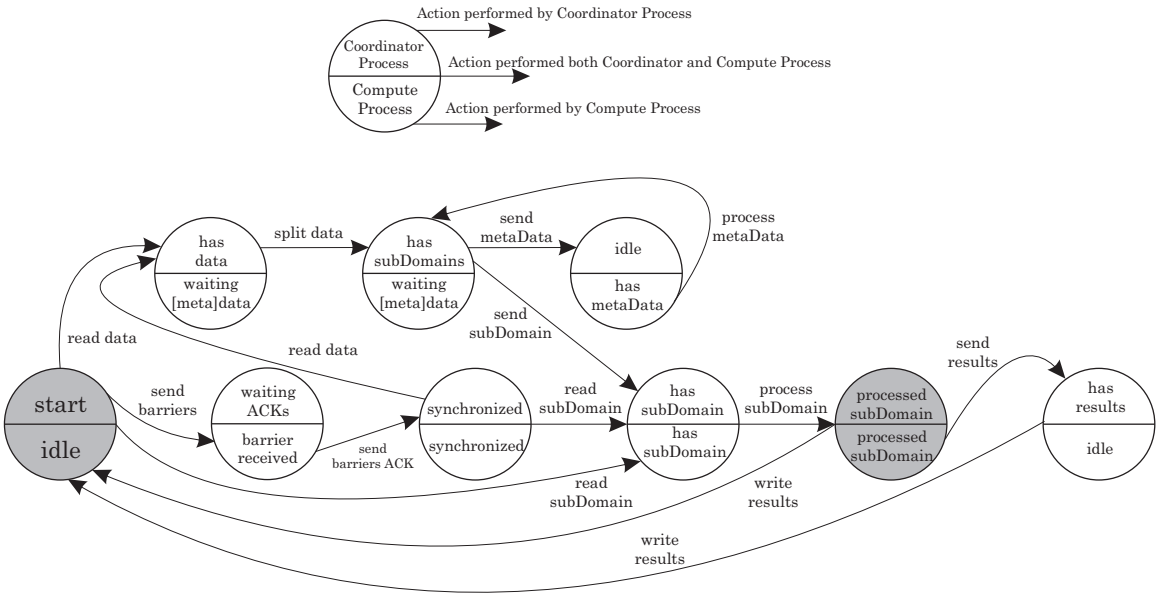


Figure 5.4: Example of application modeled using a pre-defined state graph

the compute process (bottom). Arrows starting from the top of the circle means that a coordinator process has performed the current action. Arrows starting from the bottom of the circle means that a compute process has performed the current action. When the arrow starts at the middle of the circle, then the same action has been performed by the coordinator and the compute process. Depending on the application's behavior, the graph will take the corresponding path.

The proposed technique has some advantages. The most important one is scalability. Thus, scaling the model represented by a given graph is immediate, due to it is enough with configuring the corresponding parameters like the number of processes, and the amount of data processed in each operation. Another advantage is that we can study the impact of changes in the HPC application on the overall system performance, without changing any line of code. It can be done simply by configuring the corresponding parameters with the right values. Moreover, due to SIMCAN uses an event-based environment as basis, the implementation of this technique fits much better with the SIMCAN's coding paradigm than other paradigms like procedural programming.

This technique has several important drawbacks. First, in some cases it is very difficult to obtain the behavior of any application in order to generate the corresponding graphs that model its behavior. Moreover, when the system to be modeled is a parallel or a distributed system, the complexity increases exponentially. This technique make easier to scale the desired application at the cost of reducing the accuracy.

5.2.3 Coding applications from scratch

While the former strategies for modeling applications provide an easier method for users, they are not general strategies that work for every possible case. When the desired model cannot be fitted into those strategies, the only strategy left is to build the application from scratch.

SIMCAN offers different programming schemas and a complete system API with configurable facilities, which let build any application model from scratch. Those application models could be implemented using statistical approaches (using the statistic functions provided by OMNET++) or could be implemented as a port of the Real Application with more or less detail.

This section shows several approaches for programming applications in the SIMCAN simulation platform. Depending of the fidelity degree required for simulating a given application, one of those techniques must be chosen. This degree of fidelity represent from simple schemas to a full port of the given application.

Basic schemas usually consist on representing the behavior of a system using statistical approaches. In some cases, the nature of the application is complex. Thus, trying to find statistical functions to represent the behavior of any application can be a time-consuming and difficult task. In those cases, a possible approach can be using input parameters to define the application's model. Then, each one of those parameters can be represented using a numeric value or a statistical function, depending on the model's design. For instance, some parameters can be the amount of I/O operations that application performs, the percentage of time needed for CPU processing, or the amount data sent through the network. Analytical models may provide relatively quick estimation of performance, but often lack

enough accuracy because of the difficulty in characterizing the behavior of real programs stochastically. In spite of all those disadvantages, statistical simulations can be executed quickly. Thus, this technique can be very powerful to obtain preliminary results quickly, which is very useful for exploring a wide range of simulation models and then, focuses the detailed simulation in the most important ones.

Otherwise, a port of a given application consists of translating a given application to the environment where the application will be simulated, which in this case is the SIMCAN simulation platform. Usually, when complete ports are performed, the execution of the application to be modeled and the simulation of the architecture might be interleaved [DJS94]. In this case, simulation executes the machine language instructions of the application directly on the simulation host. For instance, it can be done in order to calculate the amount of instructions to be simulated in a concrete CPU.

In summary, there are three methods for programming an application from scratch in SIMCAN, which are described in detail in following sections.

- Using the native event-programming paradigm.
- Using structured programming paradigm.
  - Using co-routines.
  - Using a pre-compiler.

5.2.3.1 Using the event-programming paradigm

The event-programming paradigm is the normal way to develop applications and modules in SIMCAN. This paradigm is specially fitted for programming simulations that have to deal with all the events that both the hardware and the software produce. The main problem is portability, due to this programming paradigm is not well suited for porting existing applications that are already implemented using structured or object-oriented programming.

This programming paradigm is completely different to the one that is normally used for structured or object-oriented programs. In general, event-driven programming is more complex than structured programming. For this reason, porting current applications to event-driven programming is a task quite complex to do by hand. Even for original applications designed purely for SIMCAN, the effort is bigger because usually the programmers lack the required skills to implement this kind of applications.

Applications developed using the event-programming paradigm are especially useful when a statistical approach is used. By the contrary, it is very difficult to port existing applications using this method. For this reason, SIMCAN also provides different methods to implement structured-programmed applications into SIMCAN.

The programming paradigm most widely used is the structured programming paradigm. Most programmers are skilled in this kind of paradigm. Moreover, most existent applications are implemented using this programming paradigm. Thus, porting them to SIMCAN becomes easier using structured programming instead of the native event-based programming.

In order to implement structured programmed applications in SIMCAN, two different approached have been proposed. First approach consists on using the co-routines provided by OMNeT++ platform and POSIX. Second approach is to build a pre-compiler that translates structured programmed applications in a event-oriented application.

5.2.3.1.1 Using co-routines

Co-routines are an easy way to implement structured-programmed applications in SIMCAN. This facility is offered by the OMNeT++ platform itself.

This is a concurrent paradigm that consists on multiple execution contexts (normally programmed in a structured way). In order to change from one context to another there must be a change of context performed by the simulation platform that is currently executed.

OMNeT++ co-routines are based on the POSIX services which let change the context between different co-routines easily. POSIX includes a set of standard functions that let us implement co-routines easily like *setcontext*, *getcontext*, *makecontext* and *swapcontext*. The main problem of this solution is that each co-routine has to define a fixed amount of memory for the stack since the beginning of the execution. This lead to a waste of memory in order to avoid stack overflows that limits the number of co-routines that can be executed at the same time.

Another drawback of this approach is that the implementation of co-routines in OMNeT++ excludes the possibility to mix it with event-driven code in the same module (a module has to be coded using event-driven programming or using co-routines). Due to this, we provide another approach in order to built structured applications.

5.2.3.1.2 Using a pre-compiler

Another approach is to implement an automatic method for transforming an application code that uses structured programming into an event-driven programming code. This approach has several advantages compared to the use of co-routines. First, the use of memory is adjusted to its real use, maximizing the number of possible executed modules. Second, the system can be configured to mix event handlers, with sequential code (by letting several events to be treated as events handlers while the rest are treated as sequential code).

Currently, we have developed a prototype of this pre-compiler, which is targeted to develop basic applications in SIMCAN using a structured programming paradigm. The main objective of this pre-compiler is to ease the task of writing applications in the SIMCAN simulation platform. This prototype has been performed using a set of pre-defined C macros. Thus, using this set of macros, users can build programs with a structured programming appearance that would be converted to a fully event-oriented program, once it is processed by the C pre-compiler.

Therefore, the basic idea is to create a reduced language based on C, which will be used for users to write simulated applications using the API (see section 3.4) provided by SIMCAN. Thus, users are able to write applications smoothly without using events, because those events are processed transparently by the new code generated by the pre-compiler.

The main difficulty of this pre-compiler lies in the synchronization of each system call

with the simulation platform. Each invocation of a system call, such an I/O operation, must wait for processing the corresponding response event. Thus, applications cannot be written using a structured programming paradigm in SIMCAN. This occurs because if two system calls are invoked sequentially, there is no warranty that the corresponding response events arrive in the same order that the sentences were invoked. Then, users must assure by hand that after invoking a new system call, the response of the previous one had been completely processed.

Therefore, in order to process completely a system call invocation is mandatory to finish the execution of the function that emulates the system call invoked. Then, once the response of this system call arrives, this function can be executed again. This can be fulfilled by creating a set of program states using the *switch* sentence. Thus, the state of the system call invoked is managed using a class attribute, whereof each time the same function is invoked, only the specific portion of code associated with the current state of the system call execution will be processed.

Moreover, local variables must store their value between consecutives invocations. Therefore, all local variables are implemented as attributes of the class that represents the simulated system call.

In order to alleviate this issue, this pre-compiler is based on generating a set of pre-defined C macros. Those macros are in charge of synchronizing each system call invoked with its corresponding response. Whereof, the generated application code can assure that all system calls are executed in the right order, by processing their corresponding event responses before invoking the next system call. Thus, users can write applications using a structured programming, and then this pre-compiler translates those applications into a set of macros for processing the corresponding events in the right order.

Listing 5.1 shows an example of an application written using the previously reduced structured language. This basic application opens a file, and then read such file completely using blocks of 1 MB. For each read block, this application performs processing operations, specifically 55000000 MIs.

Listing 5.1: Example of a simulated application in SIMCAN

```
...
int result, blockSize, i;

  // Init
  result = simcan_request_open ("file.dat");
  blockSize = 1024*1024;
  i = 0;

  // Check if the file has been successfully opened.
  if (result != 0)
    printf ("Error opening the file");
  else{

    // Read a block and process it, until reach the EOF
    while (result == 0){
    result = simcan_request_read ("file.dat", blockSize*i, blockSize);
    simcan_request_cpu (55000000));
    i=i+1;
    }
  }
...
```

Thus, the code showed in listing 5.2 is generated by using the proposed method for translating the example of sequential code (see listing 5.1) to event-driven code.

Listing 5.2: Transformation of sequential code in event-driven code

```
...
while (TRUE) {

    switch (objet.var_state) {

    case 1: send_simcan_request_open(...)
        object.var_state++; break;

    case 2: if (!received_simcan_request_open(...))
            return (0);
        objet.var_state++; break;

    case 3: objet.blockSize = 1024*1024;
        objet.i = 0;
        objet.var_state++; break;

    case 4: if (objet.result != 0)
            objet.var_state = 5;  /* goto IF (begin) */
        else
            objet.var_state = 7;  /* goto ELSE */
        break;

    case 5: printf ("Error opening the file");
        objet.var_state++; break;

    case 6: objet.var_state= 14; /* goto IF (end) */
        break;

    case 7: if  (objet. result == 0)
            objet.var_state++;  /* continue WHILE */
        else
            objet.var_state=13;  /* goto end of WHILE */
        break;

    case 8: send_simcan_request_read(...)
        objet.var_state++; break;

    case 9: if (!received_simcan_request_read (...))
            return (0);
        objet.var_state++; break;

    case 10: send_simcan_request_cpu(...)
         objet.var_state++; break;

    case 11: if (!received_simcan_request_cpu (...))
         return (0);
         objet.var_state++; break;

    case 12: i = i+1;
         objet.var_state++; break;

    case 13: objet.var_state++; break; /* end of WHILE */

    case 14: objet.var_state++; break; /* end of IF */
   }
}
...
```

Listing 5.3 shows a set of pre-defined macros, which has been written in C. Using those macros, this pre-compiler translates applications written in structured programming

to applications based on events. The main idea is that for each invoked system call, the application waits for the corresponding response event. And then, the execution of such application continues.

It has been implemented using states, which are managed using the variable *var_state*. Then, an application is divided in states, where each state represents a unique sentence in the application. Using this schema let us synchronize system calls with the simulation platform, because when a system call is invoked the application remains waiting for the corresponding event. Once the response event arrives, the application calculates the current state, and then continues with its execution.

Listing 5.3: Example of a simulated application in SIMCAN

```
/** BEGIN MACRO (start the infinite loop and the switch */
#define BEGIN bool loc_finalResult = false; \
  while (1) switch (var_state){

/** END MACRO default case and return with error */
#define END default: return (false);}

/** EXEC MACRO standard case, can handled several C++ sentences,
 * requires a state and the state var as the rest */
#define EXEC(state, sentences) case (state): \
{sentences}; \
(var_state)++; \
break;

/** IF MACRO require a bool expression and the state of the ELSE case */
#define IF(state, comment, expression, else_state) case (state):    \
if (expression) \
  (var_state)++; \
else \
  {(var_state)=(else_state);(var_state)++;}\
break;

/** ELSE MACRO requires the state of the ENDIF case */
#define ELSE(state, comment, end_state) case (state):\
(var_state)=(end_state); \
(var_state)++; \
break;

/** ENDIF MACRO not requires anything special*/
#define ENDIF(state, comment) case (state): \
(var_state)++; \
break;

/** WHILE MACRO requires a bool expression and the state of the ENDWH case */
#define WHILE(state, comment, expression, end_state) case (state):\
if (expression) \
  (var_state)++; \
else \
  {(var_state)=(end_state);(var_state)++;}\
break;

/** ENDWH MACRO requires the state of the WHILE case */
#define ENDWH(state, comment, while_state) case (state): \
(var_state)=while_state; \
break;

/** REQ MACRO requires the expression with the request function (void function) */
#define REQ(state, req_function) case (state): \
base->req_function; \
(var_state)++; \
break;
```

```
52
53 /** REQ MACRO requires the expression with the response function (bool function)
       true -> continue, false->return */
54 #define RESP(state, res_function) case (state): \
55 if (true == base->res_function) {\
56   (var_state)++; \
57   loc_finalResult = true; \
58 break; \
59 } else \
60   return (loc_finalResult);
```

When the proposed pre-compiler processes applications written using the reduced language commented before, a new C++ class is then generated. Such class represents the behavior of the simulated application using events. Therefore, the new generated class acts like a black-box for users by hiding all details for processing events. In this case, once application showed in listing 5.1 is processed by the pre-compiler, a new class is then generated. This class consists of three main parts.

First part consists of variables (see listing 5.4). Each application must contain two mandatory variables: *var_state* and *base*. First variable represent the current state of the application. Second variable is a pointer to the module that contains such application. The rest of variables are those used in the application itself (see listing 5.1), which in this case are *result*, *blockSize* and *i*.

Listing 5.4: Global variables generated by the pre-compiler

```
1
2 /** MANDATORY MEMBER contain the state of the execution (for the case statement) */
3 CodeState_T var_state;
4
5 /** MANDATORY MEMBER Associate SIMCAN_MODULE */
6 ExampleLib *base;
7
8 /** Code Variable: result -> for temporary results */
9 int result;
10
11 /** Code Variable: blockSize */
12 int blockSize;
13
14 /** Code Variable: blockSize */
15 int i;
```

Second part consists of application code. Thus, this part contains the code that represent the application showed in listing 5.1 using events. In order to accomplish this, this generated code uses the macros showed in listing 5.5.

Listing 5.5: Application code using pre-defined macros

```
1  BEGIN;
2
3  REQ (L10, simcan_request_open ("file.dat"););
4  RESP (L20, simcan_response_open (&result));
5  EXEC (L30, blockSize=1024*1024;);
6  EXEC (L40, i=0;);
7  IF (L50, if, (result != 0), L60);
8  EXEC (L50_10, base->showDebugMessage ("Error opening the file"););
9  ELSE (L60, else, L70);
10 WHILE(L60_10, while, (result == 0), L60_20);
11 REQ (L60_10_10, simcan_request_read ("file.dat", blockSize*i, blockSize););
12 RESP (L60_10_20, simcan_response_read (&result));
13 REQ  (L60_10_30, simcan_request_cpu (55000000));
```

```
14 REQ  (L60_10_40, simcan_response_cpu ());
15 EXEC (L60_10_50, i=i+1;);
16 ENDWH(L60_20, end, L60_10);
17 ENDIF(L70, end);
18
19 END;
```

And finally, the application code state (see listing 5.6).

Listing 5.6: States of compiled application

```
1
2 enum CodeState_T { L10, L20, L30, L40, L50,
3             L50_10,
4             L60,
5             L60_10,
6             L60_10_10, L60_10_20, L60_10_30, L60_10_40, L60_10_50,
7             L60_20,
8             L70
9 };
```

5.2.4 Comparative of modeling applications using SIMCAN

In this chapter have been explained three different techniques for modeling and simulating applications in SIMCAN. Each one of those techniques has its own advantages and drawbacks. The most important features of a simulation technique is the trade-off between the accuracy and CPU time needed to execute the simulation. Following is presented an overview of the previous techniques highlighting the most relevant advantages and drawbacks of each one.

Trace-driven simulations provide an easy method for reproducing the behavior of applications in a Simulated Environment. In one hand, this technique has serious practical shortcomings. First, tracing complete benchmark executions is infeasible as this requires the storage of billions of instructions. Second, simulation time is also prohibitive for such huge traces; especially if traces are used to evaluate various processor configurations for various workloads, which requires many simulation runs. Depending of the level of detail of such traces, the obtained results can be more or less accurate. In the other hand, this method is very useful for validating a model, because the behavior of a Real Application can be replayed quickly and easily in a Simulated Environment using its trace.

In conclusion, this method is useful for validating relatively small environments, but it can be impractical for large distributed environments, because tracing the complete system can be more difficult than simulating it using another technique more accurate than this one.

Second proposed technique consists on using pre-defined application models. This technique used to be less accurate than trace-driven. This technique has several important drawbacks. First, it is very difficult to obtain a model of a complex application. When the application to be modeled is a parallel or a distributed application, the complexity increases exponentially. Second, the obtained results will be not as accurate as the other simulation techniques.

Analytical models may provide relatively quick estimates of performance, but often lack sufficient accuracy because of the difficulty in characterizing the behavior of real programs stochastically. In spite of all those disadvantages, simulations using pre-defined

models can be executed faster than other techniques. Thus, this technique can be very powerful to obtain preliminary results quickly. It is very useful for exploring a wide range of simulation models and then, focuses the detailed simulation in the most important ones.

Finally, the last technique consists on writing the application from scratch. This technique combines the advantages of the previous techniques: accuracy and performance. Accuracy, because application can be ported completely to the SIMCAN simulation platform. Thus, the fidelity of the simulation application with the analogous real application is high. And performance, because this technique does not require read and translate a trace file, which can be a considerable overhead for large applications.

The main drawback of this technique is complexity, because the application has to be written completely. In some cases, porting completely applications to a simulation platform is difficult and time-consuming. Moreover, the event-based nature of the simulation platform hampers the porting process.

5.3 SIMCAN facilities for using the architectural resources

The first task to be performed in order to simulate applications in a simulated environment is to model the behavior of such applications. Basically, this behavior is defined by how this application uses the provided resources in the corresponding system where the application is executed. In SIMCAN, those resources are provided by the four basic systems described in section 3.3.

The way a simulated application uses those resources is independently of the technique used for creating the model of such application. Thus, depending of the requirements of the user, both a concrete technique and the method for using those resources must be selected. Those resources are requested from applications using the functions provided by the simulation platform, which are described in detail in section 3.4.

Figure 5.5 shows the basic schema of an application that requests the provided resources by the simulation platform. Basically, the idea consists on sending a request to the system that contains the resource requested by the application. Depending of the resource to be used, this request can be performed explicitly by the application, or by the contrary it can be performed implicitly without appearing in the application code.

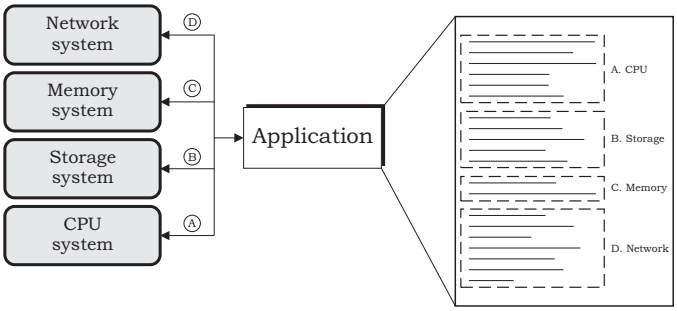


Figure 5.5: Example of resources used by a modeled application in SIMCAN

In the next sections, the methods for using those resources are described in detail.

5.3.1 SIMCAN facilities for modeling the usage of CPU resources

In order to use the computing system provided by SIMCAN, each modeled application must specify the number of instructions to be executed in the CPU. Thus, each application must use the next function provided by SIMCAN:

void *simcan_request_cpu* (long int *numInstructions*);

There are two methods for calculating the number of instructions to be simulated in the simulation platform. The first method consists on using a pre-set value configured by users. Usually, this value is calculated using statistical analysis or measuring exactly the number of instructions to be executed in each part of the code in the given application. However, this method does not require executing instructions in the real CPU where the simulation is being executed. Listing 5.7 shows a portion of code of an application modeled in SIMCAN. In this example, the function for requesting the CPU usage is invoked directly using a value of 500000000 instructions.

Listing 5.7: Example of CPU usage without using the real CPU

```
1  ...
2  simcan_request_open (file);
3
4  simcan_request_read (file, 0, 1024);
5
6  simcan_request_cpu (500000000);
7
8  simcan_request_write (file, 0, 1024);
9
10 simcan_request_close (file);
11 ...
```

Second method consists on executing a concrete portion of code that contains computing operations of a given application, in the same CPU where the simulation is being executed. The main disadvantage of this method is that the simulated CPU must be the same as the CPU used for executing the simulation. Then, the number of instructions performed in that portion of code in the application *app*, executed in CPU is calculated using the equation 5.4.

$$numInstruction(app, CPU) = timeCPU_{real} \cdot (CPU_{MIPS} \cdot 10^6) \tag{5.4}$$

where $timeCPU_{real}$ is the time needed for executing the portion of code, and CPU_{MIPS} is the speed of the processor (measured in MIPS) where that code is executed. Then, the executed code is, more or less, the real code working with the real data. Thus, the simulated part comes from the simulated architecture. In this context, the execution time of the application is calculated from the real execution time on the CPU where the simulation is being held.

This accounting of instructions can be performed both explicitly and implicitly. First alternative requires measuring explicitly the time required for executing the corresponding computing operations, in the application to be modeled. Then, the user that codes the application must measure the time needed for executing each portion of code to be simulated in the CPU system provided by SIMCAN. Listing 5.8 shows an example of this alternative. In this simulated application, the time spent for executing the loop is measured using two

time-stamps, at the beginning and the end of the loop. Then, the number of instructions are calculated using equation 5.4. Finally, the function provided by the API is invoked using the calculated value.

Listing 5.8: Example of CPU usage calculating explicitly the number of instructions

```
...
simcan_request_open (file);

simcan_request_read (file, 0, 1024);

t1 = time();

for (i=0; i<1000; i++)
  result = result + (i + sqrt(i) * 2);

t2 = time();

numInstructions = (t2-t1) * CPUmips * pow (10,6);

simcan_request_cpu (numInstructions);

simcan_request_write (file, 0, 1024);

simcan_request_close (file);
...
```

Second alternative consists on calculating this amount of time in the API module. Thus, the process of accounting the number of instructions is hidden to the application. Basically, the idea consists on measuring the time spent between two blocking functions. Then, users don't have to measure the number of instructions to be simulated, because it is implicitly calculated by the rest of functions in the API module. Listing 5.9 shows an example of this alternative.

Listing 5.9: Example of CPU usage calculating implicitly the number of instructions

```
void blockOperation (){

  t2 = time();

  numInstructions = (t2-t1) * CPUmips * pow (10,6);

  if (numInstructions > 0)
    simcan_request_cpu (numInstructions);

  // Implementation of the corresponding function
  ...
  ...

  t1 = time();
}
```

This schema works well for simulating architectures with one or more CPUs. Also it is valid for simulating CPU schedulers and for performing light simulations of different CPUs just by obtaining the performance ratio between the real CPU and the simulated CPU.

5.3.2 SIMCAN facilities for modeling the usage of memory resources

The amount of memory needed for applications simulated in SIMCAN are categorized in four groups. Those groups, which are described in detail in section 3.3.2, consist of:

1. Code.
2. Global variables.
3. Local variables.
4. Dynamic variables.

The amounts of memory managed for each one of the purposes previously listed are measured by SIMCAN, using the functions provided by the corresponding API (see listing 3.2). This approach is similar to the CPU accounting explained before. Basically this approach consists on measuring the maximum memory used by the application, and then, requesting this amount of memory using the functions provided by the API. This API basically provides two functions for managing memory:

void *simcan_request_allocMemory* (int *memorySize*, int *region*);

void *simcan_request_freeMemory* (int *memorySize*, int *region*);

Although SIMCAN provides mechanism for accounting the memory needed for each one of the previous purposes, this resources must be explicitly required by the user. Following are explained the approaches for accounting each area of memory.

In order to perform the accounting of the memory needed for the code and global variables, several calls to the function of allocating memory are needed. This value is a constant during the whole execution. Thus, the user has to estimate the corresponding amount of memory before the execution (by hand, using an automatic preprocessing or during the compilation of the original application if this is possible). Then, one call is needed for allocating the memory needed for the code, and other one is needed for allocating the amount of memory needed for global variables. Listing 5.10 shows an example for measuring this memory.

Listing 5.10: Example of managing memory for code and global variables

```
1  void mainApplicationFunction(){
2
3    simcan_request_allocMemory (128000, REGION_CODE);
4    simcan_request_allocMemory (2048, REGION_GLOBALVAR);
5
6    // Core of the simulated application
7    ...
8    ...
9
10   simcan_request_freeMemory (128000, REGION_CODE);
11   simcan_request_freeMemory (2048, REGION_GLOBALVAR);
12 }
```

The approach for measuring the memory needed for local variables is the most difficult and complex. Each time a function is invoked, a copy of their local variables is reserved, and every time a function ends, its local variables are freed. The method to perform this accounting is by tagging on the code the amount of memory required for the local variables of each function. This value has to be estimated by the user. Then, the simulation platform must process the needed amount of memory each time a function is invoked in order to increase the use of memory accordingly. Thus, each time a function ends, the simulation

platform has to reduce the use of memory. Listing 5.11 shows an example of measuring this memory.

Listing 5.11: Example of managing memory for local variables

```
void mainApplicationFunction(){
  ...
  localVariableExample();
  ...
}

void localVariableExample(){

  int size, result, _localMem;
  char array[10];

    _localMem = (2 * sizeof(int)) + (10 * sizeof(char));

    simcan_request_allocMemory ( _localMem, REGION_LOCALVAR);

    // Core of function localVariableExample
    ...
    ...

    simcan_request_freeMemory ( _localMem, REGION_LOCALVAR);
}
```

Finally, measuring dynamic memory is the simplest approach. In order to accomplish this task, it is enough to invoke the memory functions for allocating and freeing memory in order to perform the accounting. Listing 5.12 shows an example of measuring this memory.

Listing 5.12: Example of managing memory for dynamic variables

```
void mainApplicationFunction(){
  ...
  simcan_request_allocMemory (1024*1024*10, REGION_DYNAMICVAR);
  ...
  simcan_request_freeMemory (1024*1024*10, REGION_DYNAMICVAR);
  ...
}
```

5.3.3 SIMCAN facilities for modeling the usage of storage resources

The storage system in SIMCAN is in charge of managing accesses to data. Thus, the objective of this system is two-fold. First, this system must calculate the time needed for performing each provided service. Second, this system must let simulated applications manage real data.

The greatest difference between the execution of a Real Application and the execution of a Simulated Application is the execution of the system services. In simulation, those services are done by underlying the simulated architecture. This simulated architecture is mainly in charge of calculating the time spent that should correspond with those services in a Real Environment. But in most cases, those services don't perform the real service, which involves real data. The problem is that the data that the service is supposed to return is required in order to continue the execution, when the application is simulated in full detail. This is the reason why simulations require that the system services also deals with real data.

This doesn't mean that the simulation platform has to mimic the complete resources that are being simulated. In contrast, it is a better choice using the services of the system where the environment is being simulated in order to obtain the data required. Satisfying this requirement is easy for components like the file system (just by using the file system of the host operating system) or storage devices like disk drives. The performance of the host system is not important because the time required for the system services is already being calculated by the simulated architecture.

The problem of this alternative is that the system where the simulation is being executed has to provide the required services. Otherwise, simulation cannot perform the services required completely because of lack of resources regarded with the data.

SIMCAN provides those two alternatives for using the storage system in each simulated application. Depending of the level of detail required for simulating applications, the user must chose whether calculating the times for performing the corresponding operations is enough. Or by the contrary, the simulation application must deal with real data.

In the case the application to be simulated requires the use of real data, the first task to be performed by users is to configure correctly the system where the simulation will be executed, because the user must assure that such system will provide the required services.

Currently, the API of SIMCAN provides a set of services in order to manage files. Those services (see listing 3.3) correspond with the operations for open, close, create and delete files in the storage system. Moreover, functions for read and write data in the set of files located in the storage system are also provided. Those services might be configured in order to manage real data provided by an external system from the simulation platform.

Listing 5.13 shows a portion of a simulated application in SIMCAN. This application opens a file, reads 1 MB using blocks of 1 KB, and closes the file. Listing 5.14 shows the implementation of a block operation. In this case, when flag USE_REAL_DATA is active, the accounting is performed implicitly. Otherwise, this accounting must be performed explicitly. Note that the time calculated in the simulation is the same for both alternatives, because calls of the system where the simulation is executed (open, read and close) does not have any effect on the simulated time.

Listing 5.13: Portion of a simulated application using the storage system API

```
...
int readBytes, i;
void* realData;

    readBytes = 0;
    simcan_request_open (file);

    while (readBytes < (1024*1024)){
      realData = simcan_request_read (file, readBytes, 1024);
      readBytes += 1024;
    }

    simcan_request_close (file)
...
```

Listing 5.14: Portion of a simulated application using the storage system API and real data

```
void* simcan_request_read (char* fileName, unsigned int offset, unsigned int size){

  void* realData;
  ...
```

```
5
6       if (USE_REAL_DATA){
7         fd = open (fileName);
8         lseek (fd, offset, SEEK_SET);
9         read (fd, realData, size);
10        close (fd);
11      }
12      else
13        realData = NULL;
14
15      // Core of the function simcan_request_read
16      ...
17      ...
18
19    return realData;
20  }
```

5.3.4 SIMCAN facilities for modeling the usage of networking resources

The network system is in charge of managing communications between different applications in a simulated environment. Those applications can be hosted in the same node, or by the contrary, in different nodes. The services provided by this simulation platform (see listing 3.4) include:

- Listen for an incoming connection.
- Establish a remote connection with a remote application.
- Send data to remote applications.
- Receive data from remote applications.

Those services are totally independent of the strategy used for simulating the network. Thus, using the same API, several strategies can be used for simulating the network of a distributed system. The difference between using one or other strategy is the level of accuracy obtained and the time needed for executing the simulation. In general, the more detailed the strategy used, the slower the execution of the simulation.

As occurs with the storage system, some applications require using real data through the network system in order to be simulated. However, using real data in the network system presents more drawbacks than the storage system. First, a set of simulated applications are executed in the same machine, whereof the same real network interface, and the same real IP address must be shared by all simulated applications. Moreover, in the case of using parallel simulations, a set of real machines must be organized for sharing its network resources by the simulated applications spread among real nodes. Those reasons make this approach unfeasible.

Currently, in order to use real in the network system, each message processed by this system must store the corresponding data to be sent to the corresponding component. This solution entails several drawbacks. First, the simulation is much slower. Second, the amount of memory needed is higher.

5.4 Summary

In this chapter have been described three different techniques for modeling applications in the SIMCAN simulation platform. Each technique is targeted to achieve a concrete purpose.

Trace-driven technique is very useful for validating relatively small environments. Moreover, this technique does not require a great effort to be fulfilled. Thus, applications can be quickly modeled using this technique. The main drawback of this technique is scalability, because tracing applications executed in large environments will generate huge trace files, making impractical this solution.

Using generic models for modeling applications is a useful approach for modeling applications with a well-known behavior. The advantage of this method is that several applications can be easily simulated by configuring a graph with the required parameters. Otherwise, the drawbacks that this technique presents are mainly the difficulty for obtaining the behavior of a given application.

Last technique consists on coding the application from scratch in the SIMCAN simulation platform. The main advantage of this technique is flexibility. Each application can be modeled from a very basic schema to a complete port of such application. The main drawback of this technique is the effort and time required for coding the required application.

Moreover, several methods for using the resources provided by each one of the basic systems modeled in SIMCAN have been described in detail. Those resources can be used equally for applications modeled using each one of the previously described techniques.

Chapter 6

Validation of the SIMCAN simulation platform

This chapter shows the process used for validating the SIMCAN simulation platform. The main objective of this chapter is to demonstrate that SIMCAN can accurately model and simulate both real environments and applications, obtaining coherent and consistent results.

Thus, in order to demonstrate the accuracy of this simulation platform, several real environments and a set of well-known applications have been modeled using SIMCAN. Therefore, those applications are executed in both real and the modeled environments. Finally, results obtained in real systems and results obtained in simulated environments are compared.

6.1 Validation process overview

Model validation is usually defined to mean "substantiation that a computerized model within its domain of applicability possesses a satisfactory range of accuracy consistent with the intended application of the model" [S.79].

After a simulator has been developed, implemented, and debugged, it must be tested for correctness and accuracy. Performance model validation involves generating test cases, stimulating the model under test, and comparing execution results to a known reference. Also, all validation methods require a reference to determine if a test sequence passes or fails.

Determining that a simulator is absolutely valid over the complete domain of its whole intended field of applicability is a very hard and time-consuming task. Thus, the level of accuracy of a given simulator can't be calculated for the entire domain this simulator is targeted using a single value, because this accuracy depends directly of the system to be modeled. Instead, a model is considered valid for its intended application if tests and evaluations performed are conducted until sufficient confidence [G84]. Otherwise, a model is considered invalid if corresponding tests show that the model does not have the sufficient accuracy for a set of experimental conditions.

However, absolute accuracy is not always strictly necessary, and in many cases it is

not even desired, due to its high engineering cost and the great amount of time required for executing simulations. In many situations, substituting absolute with relative accuracy between different timing simulations is enough for users to discover trends for the proposed techniques.

The main objective of this chapter is to demonstrate that SIMCAN is able to model and simulate accurately Real Environments by obtaining coherent and consistent results, even when architectural and configuration changes are made. Most of all, we intent to demonstrate that results obtained when concrete changes are applied to the real system, have the same impact on the overall system performance than those obtained when the same changes are applied to the model.

Currently in literature there are various validation techniques which are used for verifying and validating both the submodels and the overall model. Some of those techniques are *comparison to other models*, *degenerate tests*, *event validity*, *extreme condition tests*, and *face validity* [Sar05]. Those techniques can be used either subjectively or objectively. A technique is considered objective when some type of statistical test or mathematical procedure is applied, like a hypothesis tests or confidence intervals. Otherwise, a technique is considered subjective when the decision whether a model is valid or not is made based on the results of the various experiments and evaluations.

A simulation model should only be developed for a set of well-defined objectives. Thus, a model will be valid or not depending of the user's requirements. In order to obtain a high degree of confidence in a model and its results, the comparison between the model's and system's input-output behaviors for at least two different sets of experimental conditions is usually required [Sar05].

In this chapter we propose a strategy for performing the validation process of the SIMCAN simulation platform. The basic idea of this strategy is to execute a set of Real Applications in different Real Environments and then, compare the obtained results with the corresponding simulated ones. Thus, both absolute and relative results will be studied, in order to analyze the tendency and behavior of each experiment executed in the corresponding environments. Those experiments are designed for checking a wide range of architectural designs and configurations which include different architectures of switches for connecting the nodes in a network, executing applications using different number of processes and using different storage configurations. This validation process includes both objective and subjective techniques.

The objective technique consists on calculating both the error ratio and the correlation between the real system and the models. The error ratio is expressed for the 95% confidence interval and calculates how accurate is the model compared with the real system. Thus, the closer to 0 is the error ratio, the more accurate is the simulation.

Correlation shows the strength and the direction of the relationship between two random variables. This correlation provides a value that shows how strong the level of dependency of two lineal variables is. In this process, the Pearson's correlation coefficient will be calculated for this purpose. This coefficient is a value between +1 and -1 (inclusive) that shows the linear dependence between two random variables, which in this case are the results obtained in Real and Simulated Environments.

Then, for two series of n measurements of X and Y written as x_i and y_i, where i = 1, 2, ..., n, then the sample correlation coefficient can be used to estimate the population

Pearson correlation r between X and Y as shows equation 6.1.

$$r_{xy} = \frac{\sum_{i=1}^{n}(x_i - \bar{x})(y_i - \bar{y})}{(n-1)s_x s_y} \tag{6.1}$$

where $\bar{x}$ and $\bar{y}$ are the sample means of X and Y, s_x and s_y are the sample standard deviations of X and Y. The value of this coefficient means:

- If (r=0) there is no lineal relationship between the random variables.
- If (r=1) there is perfect positive correlation.
- If (r=-1) there is perfect negative correlation, which means that when one variable increases, the other decreases equally.
- If ($0 < r < 1$) there is a positive correlation. The closer to 1 is the value of r, the better is the correlation.

Subjective techniques used in this process consist on creating charts with the results obtained in the execution of the experiments in both Real and Simulated Environments, and then analyze and compare those results in order to take a decision about validating or rejecting the model.

A model will be considered valid for a set of experimental conditions if the model's accuracy is within its acceptable range, which is the amount of accuracy required for the model's intended purpose. Our results demonstrate the effectiveness of the proposed validation method at predicting the performance of a concrete system. In summary, the validation process performed in this chapter consists of a set of steps listed as follows:

1. A set of Real Environments are chosen to perform the validation process.
2. A set of Real Applications are chosen to be executed in Real Environments (1).
3. Each Real Application (2) is executed in each one of the Real Environments (1).
4. Each Real Environment (1) is modeled using the SIMCAN simulation platform.
5. Each Real Application (2) is modeled using the SIMCAN simulation platform.
6. Each Simulated Application (5) is executed in the corresponding Simulated Environment (4).
7. Statistical analysis is performed with the data sets obtained as results (3 and 6).
8. Finally, results obtained from the statistical analysis (7) are studied both objectively and subjectively in order to validate the model, and to obtain the corresponding conclusions.

6.2 Environments used for the validation process

In order to perform the validation process of the SIMCAN simulation platform, two different environments have been chosen for executing the corresponding experiments.

First environment is a PC network architecture. This environment consists of 22 nodes and 1 switch. Those nodes are connected with the switch using an Ethernet Gigabit communication network. Each one of those nodes uses an independent Ethernet cable which is connected to the switch. Figure 6.1 shows a basic schema of this environment. Table 6.1 shows a detailed specification of this architecture. In the rest of this chapter this environment will be referenced as *environment_ 1*.

Number of nodes: 22
CPU Intel(R) Xeon(R) CPU E5405 @ 2.00GHz
4 GB of RAM memory
1 TB of Storage
Operating System Linux Debian. Kernel version 2.6.26-2-686
Network Ethernet Gigabit

Table 6.1: Detailed features of *environment_ 1*

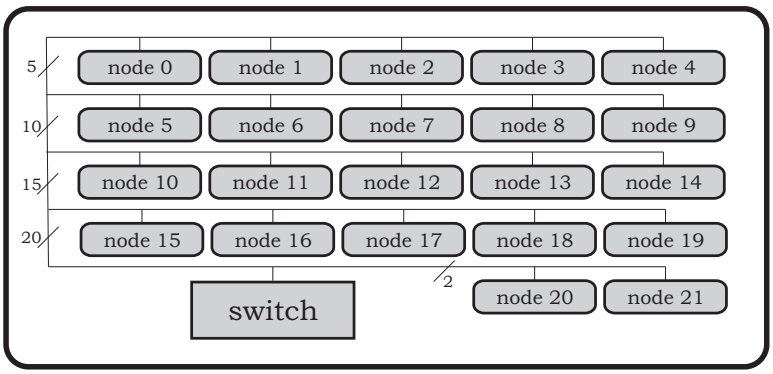


Figure 6.1: Basic schema of *environment_ 1*

Second environment is a PC network architecture with two levels of switches. This environment consists of 42 nodes and 3 switches. Those nodes are connected with the corresponding switch using an Ethernet 10/100 communication network. Each one of those nodes uses an independent Ethernet cable, which is connected to the corresponding switch. Figure 6.2 shows a basic schema of this environment. Table 6.2 shows a detailed specification of this architecture. In the rest of this chapter this environment will be referenced as *environment_ 2*.

Number of nodes: 42
CPU AMD Athlon(tm) 64 X2 Dual Core, 2.2 GHz
2GB of RAM memory
160 GB of Storage 5400 rpm
Operating System Linux Debian. Kernel version 2.6.30-2-686
Network inside classes: Ethernet 10/100
Network outside classes:Ethernet Gigabit

Table 6.2: Detailed features of *environment_ 2*

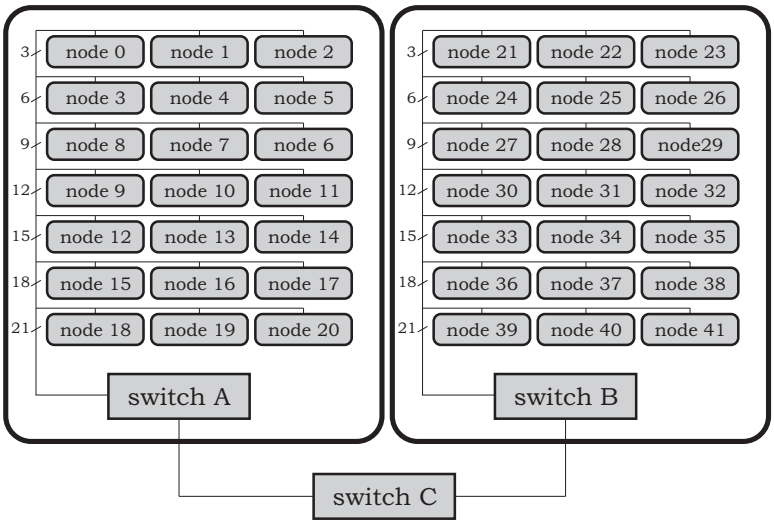


Figure 6.2: Basic schema of *environment_ 2*

6.3 Applications used for the validation process

In High Performance Computing (HPC) platforms, both network communication and I/O operations are a key factor that determines the performance. Thus, the set of applications selected to perform the validation process is focused in those systems. In order to check the accuracy of the simulation platform, three applications will be executed in both environments: I/O benchmark called IOZone [WD05], MPI benchmark called mppTest [GL99] and a parallel application called BIPS3D [FSI+07].

6.3.1 IOZone benchmark

In order to analyze the accuracy and correctness of the storage system provided by SIMCAN, a benchmark for file systems called IOZone has been chosen. This benchmark generates and measures a variety of file operations. Moreover, IOZone is useful for determining a broad files system analysis of a vendor's computer platform.

6.3.2 mppTest benchmark

The application mppTest can be used to quickly characterize the performance of an MPI implementation in a variety of ways. Using this benchmark, both collective and non-collective experiments have been performed. Using this benchmark suite we intent to measure the IPC (Inter Process Communication) performance. Over the years, the MPICH group [GLDS96] has developed this suite of programs that characterizes the performance of a message-passing environment. Using this application we intent to check the accuracy of both the INET framework and the implementation of the corresponding MPI calls in the SIMCAN simulation platform.

6.3.3 BIPS3D

Finally, a real application that mixes communication between processes and performs massively I/O operations has been used. This application is called BIPS3D.

BIPS3D is a 3-dimensional simulator of BJT and HBT bipolar devices [ALP03]. The goal of the 3D simulation is to relate electrical characteristics of the device with its physical and geometrical parameters. The basic equations to be solved are Poisson's equation and electron and hole continuity in a stationary state. Finite element methods are applied in order to discretize the Poisson equation, hole and electron continuity equations by using tetrahedral elements, as shown Figure 6.3. The result is an unstructured mesh.

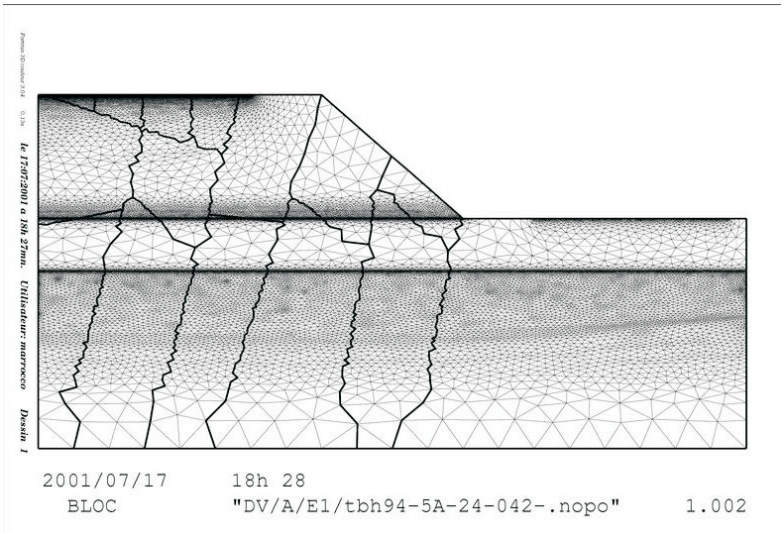


Figure 6.3: Discretization of a device.

Using the METIS library [KK98], this mesh is divided into sub-domains, in such a manner that one sub-domain corresponds to one process, as shown in Figure 6.4. In this Figure, we can observe the mesh division into 7 sub-domains (one sub-domain per color). The next step is decoupling the Poisson equation from the hole and electron continuity equations. They are linearized by Newton method. Then, for each sub-domain in a parallel manner, the part corresponding to the associated linear system is constructed. Each system is solved using domain decomposition methods. Finally, the results are written to a file.

The initial BIPS3D version, the results were gathered at a root node, which stores the

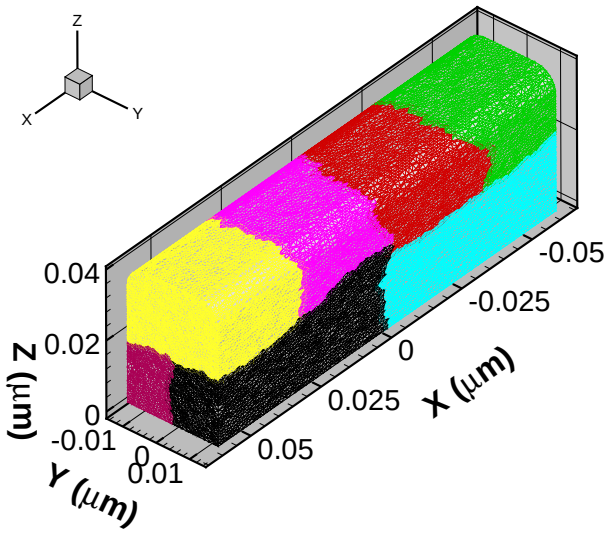


Figure 6.4: 3-dimensional simulation.

data sequentially to the file system. A modified version of BIPS3D to use collective writes during the I/O phase has been implemented by Filgueira et al [FSI+07]. In the parallel I/O BIPS3D version, each compute node uses the distribution information initially obtained from METIS and constructs a view over the file. The view is based on an MPI data type.

In this section, both the regular BIPS3D application and the modified one for performing parallel I/O have been used.

6.4 Validation process

This section describes the results obtained in the execution of the previous applications in both Real and Simulated Environments, in order to check the correctness and accuracy of the SIMCAN simulation platform. Those experiments are grouped in three sections, depending of the targeted system to be analyzed. Firstly, the storage system will be analyzed by executing the IOZone benchmark. Next, networking and Inter Process Communications (IPC) will be analyzed by executing the mppTest benchmark suite. Finally, a parallel application called BIPS3D, that mixes both massive I/O operations and communication between processes will be executed, with the purpose of analyzing both the storage and network systems.

Both error ratio and Pearson's coefficient calculated in those tests are shown in table 6.5 and table 6.6.

6.4.1 Storage system experiments

The IOZone application has been used for checking the accuracy of the storage system provided by the SIMCAN simulation platform. Each experiment has been executed in both *environment_1* and *environment_2* using two different configurations. First, the IOZone is executed using a local storage system. Second, IOZone is executed using a remote storage system by mounting a partition through NFS. Figure 6.5 shows the configuration for the

execution of IOZone in *environment_ 1*. Similarly, figure 6.6 shows the configuration for the execution of IOZone in *environment_ 2*.

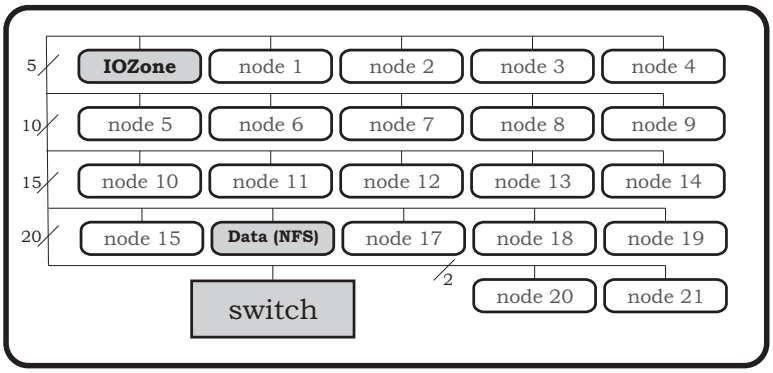


Figure 6.5: Configuration for executing IOZone in *environment_ 1*

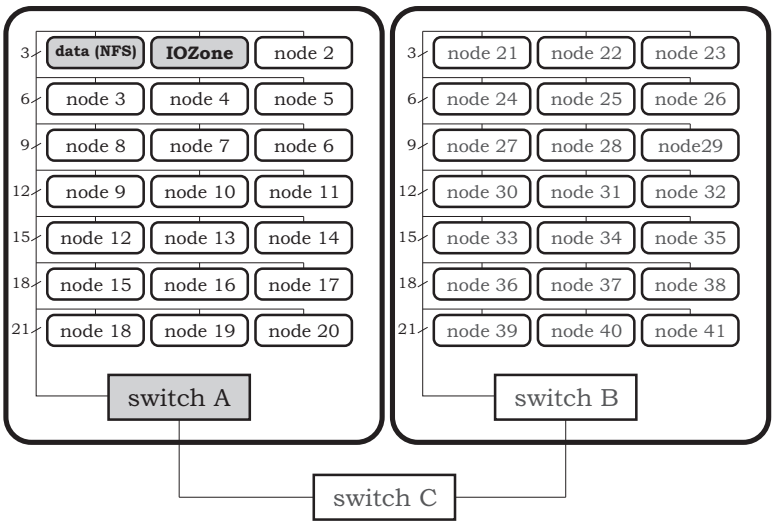


Figure 6.6: Configuration for executing IOZone in *environment_ 2*

Node labeled with "**IOZone**" represents the node where the application is executed. In the executions that use local storage, I/O operations are performed locally in this node. Otherwise, node labeled with "**Data (NFS)**" represents the node that contains the storage system where the remote partition is mounted. Thus, I/O operations are performed remotely using NFS. Nodes with white background are not used in those experiments.

Each IOZone execution performs a total of 130.942 operations which includes: read, write, re-read, re-write, read backwards, random read/write, open and create. A total of 18GB of data is read and written in each execution. Each experiment has been executed 30 times in each environment for each configuration.

The Pearson's coefficient has not been calculated in those experiments, because the results obtained in this benchmark are not enough for showing a tendency that can be compared for each one of those cases.

Figures 6.7 and 6.8 show the results (average time of 30 executions) of the IOZone

benchmark in *environment_ 1* and *environment_ 2* respectively. Dark grey columns represent the results obtained in the execution of IOZone in a Real Environment. Otherwise, light grey columns represent the results obtained in the execution of the same benchmark in a Simulated Environment.

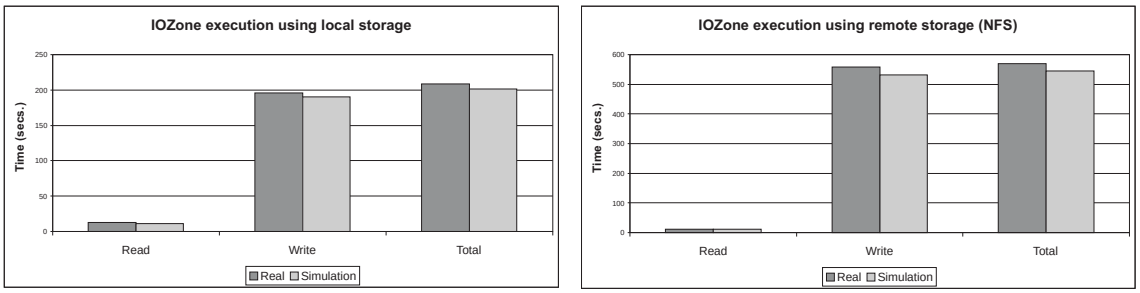


Figure 6.7: Execution of IOZone in *environment_ 1*

The charts of figure 6.7 show the results of IOZone using both local and remote storage configurations in *environment_ 1*. Those charts (see figure 6.7) show that the experiments executed in a Real Environment using remote storage, have a slowdown factor of 2.7 compared with the same experiments using local storage. This difference of performance is mainly caused due to the network is a bottleneck when I/O operations are performed remotely. This tendency is also reflected in the results achieved in simulations, which obtain the same slowdown factor of 2.7.

Table 6.3 shows the average times, measured in seconds, of the IOZone execution in *environment_ 1*. The error ratio obtained in the simulation of this experiment is 3.36 % ± 0.52 for local storage, and 4.39 % ± 0.87 for remote storage.

I/O operation / Experiment	Read	Write	Open	Close	Create	Total
Local - Real env.	12.42	196	0.00059	0.000476	0.000462	208.42
Local - Simulated env.	11.02	190.44	0.000576	0.000486	0.00045	201.46
Remote - Real env.	12.09	556.47	0.000488	0.001355	0.00043	568.58
Remote - Simulated env.	11.7	532.45	0.000628	0.000512	0.00045	544.15

Table 6.3: Average times (in seconds) of IOZone execution in *environment_ 1*

The charts of figure 6.8 show the results of IOZone using both local and remote storage configurations in *environment_ 2*. The execution of IOZone in *environment_ 2* shows that the network causes a very important impact on the overall benchmark performance. In this case, Real Application using remote storage has a slowdown of 9.8 compared with the execution of the same experiments using local storage. This drop of performance is caused by the slow network used. In those experiments, simulation shows almost the same tendency than the real system, obtaining in this case a slowdown factor of 9.46.

Table 6.4 shows the average times, measured in seconds, of the IOZone execution in *environment_ 2*. The error ratio obtained in the simulation of this experiment is 4.24 % ± 1.4 for local storage, and 7.38 % ± 1.79 for remote storage.

In general, the error ratio obtained in those experiments is low (see table 6.5 and table

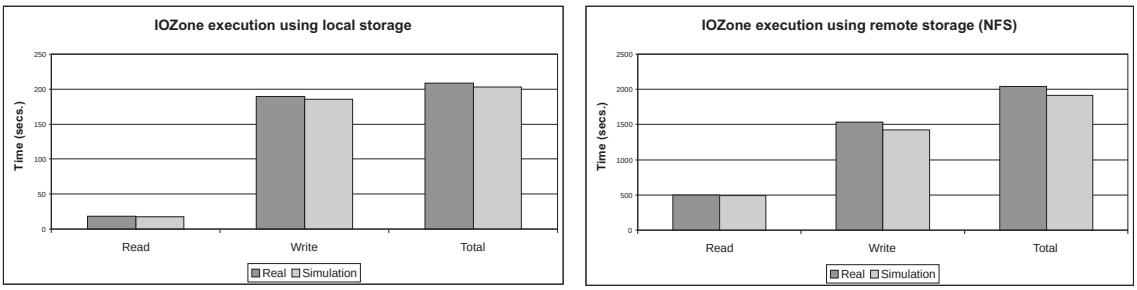


Figure 6.8: Execution of IOZone in *environment_2*

Experiment \ I/O operation	Read	Write	Open	Close	Create	Total
Local - Real env.	18.36	190.38	0.000878	0.004084	0.000532	208.75
Local - Simulated env.	17.2	185.56	0.000576	0.000486	0.00045	202.76
Remote - Real env.	516.69	1532.45	0.01472	0.000731	0.009296	2049.17
Remote - Simulated env.	489.34	1423.43	0.000628	0.000512	0.00051	1912.43

Table 6.4: Average times (in seconds) of IOZone execution in *environment_2*

6.6), even when several environments and different configurations are used for simulating the same benchmarks, which requires different components for modeling each environment like disk drives, communication networks, etc. Moreover, the slowdown factor obtained between the experiments that use local storage and experiments that use remote storage, are practically identical in both Real and Simulated Environments.

However, the error ratio obtained in those experiments that use remote storage is greater than the error ratio obtained when local storage is used. This is because local experiments only use the storage system, and experiments that use remote storage require the use of both storage and network systems, which generates an accumulative error of both systems.

6.4.2 Process communication experiments

This section describes the mppTest benchmarks executed in order to check the accuracy of the network system and Inter Process Communications (IPC). Basically, two different benchmarks of the mppTest suite have been used: Round-Trip and Broadcast. Each one of those experiments has been executed 30 times.

First benchmark, called Round-Trip, uses non-collective operations. Basically, this benchmark consists on sending and receiving messages (ping-pong) using different message sizes. Each operation always involves a pair of processes. This benchmark has been performed using several processes and different message sizes.

Second benchmark, called Broadcast, uses collective operations. This benchmark consists on synchronizing processes and broadcast messages among the processes involved in the execution of this benchmark. Several processes and different message sizes have been used for executing this benchmark.

Each one of those benchmarks has been executed in both *environment_1* and *envi-*

ronment_ 2. Figure 6.9 shows the configuration for the execution of mpptTest in *environment_ 1*.

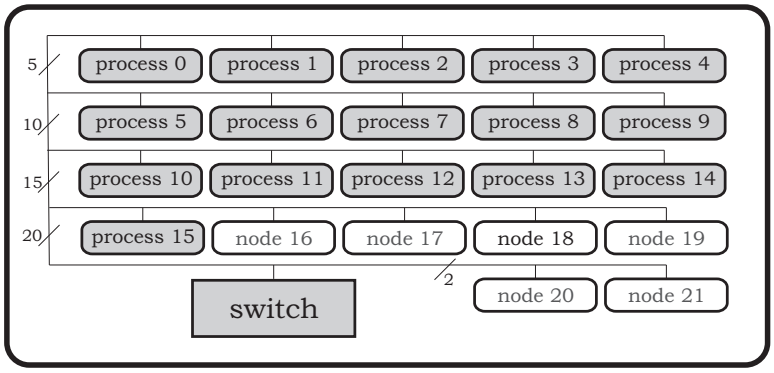


Figure 6.9: Configuration for executing mppTest in *environment_ 1*

Nodes labeled with "Process X" represent nodes where a MPI process is executed. Nodes with white background are not used in those experiments. In each one of those experiments, maximum time, minimum time and average time for a concrete message size used is shown.

Figure 6.10 shows the results obtained in the execution of the Round-Trip benchmark in *environment_ 1* using 2 processes. When message sizes from 10 KB to 100 KB are used (left chart), an error ratio of 7.2 % ± 0.6 is obtained. The Pearson's correlation coefficient is 0.9842. In those experiments that use message sizes from 100 KB to 1 MB (right chart) the error ratio obtained is 10.2 % ± 0.9, and the Pearson's correlation coefficient is 0.9809.

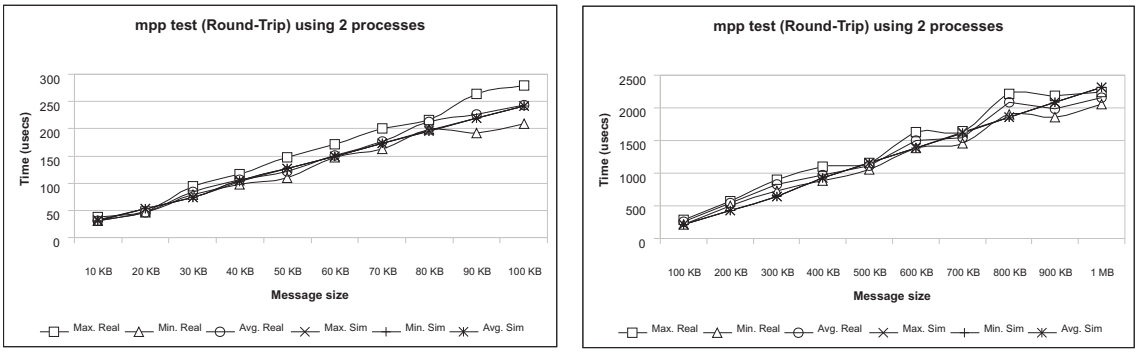


Figure 6.10: Execution of mppTest (Round-Trip) in *environment_ 1* using 2 processes

Figure 6.11 shows the results obtained in the execution of the Round-Trip benchmark in *environment_ 1* using 16 processes. When message sizes from 10 KB to 100 KB are used (left chart), an error ratio of 7.7 % ± 0.6 is obtained. The Pearson's correlation coefficient is 0.9827. In those experiments that use message sizes from 100 KB to 1 MB (right chart) the error ratio obtained is 8.9 % ± 0.8, and the Pearson's correlation coefficient is 0.9852.

The results obtained when 2 and 16 processes are used (see figures 6.10 and 6.11) are very similar. This similarity is obtained due to this experiment uses a pair of processes for performing the corresponding communications, whereof the architecture of switches used does not suffer congestion when the number of processes increases, because involved

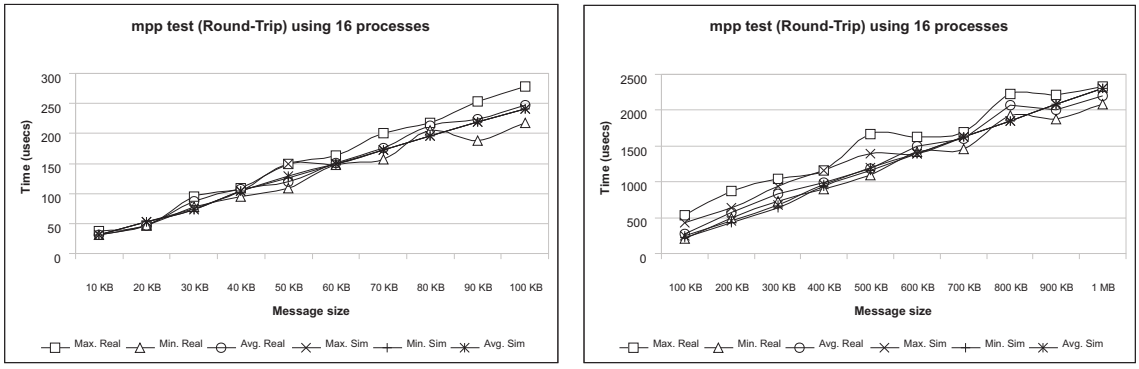


Figure 6.11: Execution of mppTest (Round-Trip) in *environment_ 1* using 16 processes

processes do not share the communication channels. This behavior is obtained in both Real en Simulated Environments. Those charts show that in almost all cases, results obtained in simulation are placed between the maximum and the minimum results obtained in the execution of the Real Applications.

In general, experiments executed in Real Environments reflect more variability than the analogous models. This is caused due to the influence of several variable elements that are present in real systems, such as operating system latencies or external noise, which affects the performance of the experiment. Otherwise, those elements are not modeled in the proposed simulation platform. However, the objective of this simulation platform is not to reflect that variability caused by additional elements, but estimating the overall system performance.

Moreover, all values of the Pearson's correlation coefficient calculated in those experiments are very close to 1, which means a very strong correlation between the Real Environment and the model. This demonstrates that the behavior of experiments executed in Real Environments when some changes are applied, like the message size and the number of processes, is reflected equally in the model.

Figures 6.12 and 6.13 show the results obtained in the execution of the Broadcast benchmark in *environment_ 1* using 2, 4, 8, and 16 processes. Message sizes used go from 10 KB to 100 KB. The error ratio obtained in those experiments is 9.9 % $\pm$ 1.1, 11.9 % $\pm$ 1.6, 11.3 % $\pm$ 0.9, 10.8 % $\pm$ 1.0 for 2, 4, 8, and 16 processes respectively. The Pearson's correlation coefficient is 0.9762, 0.9905, 0.9805 and 0.9799 for 2, 4, 8, and 16 processes respectively.

The charts of figures 6.12 and 6.13 show that results obtained in simulations are very close to results obtained in the Real Environment. There are few cases where results obtained in simulation are not placed between the maximum and the minimum results obtained in the execution of the Real Application. This is mainly caused by several peaks obtained in real executions. Besides the tendency of simulation and real execution is the same, simulation follows a linear behavior, while the execution of the Real Application has several peaks.

Figure 6.14 shows the configuration for the execution of mppTest in *environment_ 2*.

Figure 6.15 shows the results obtained in the execution of the Round-Trip benchmark in *environment_ 2* using 2 processes. When message sizes from 10 KB to 100 KB are used

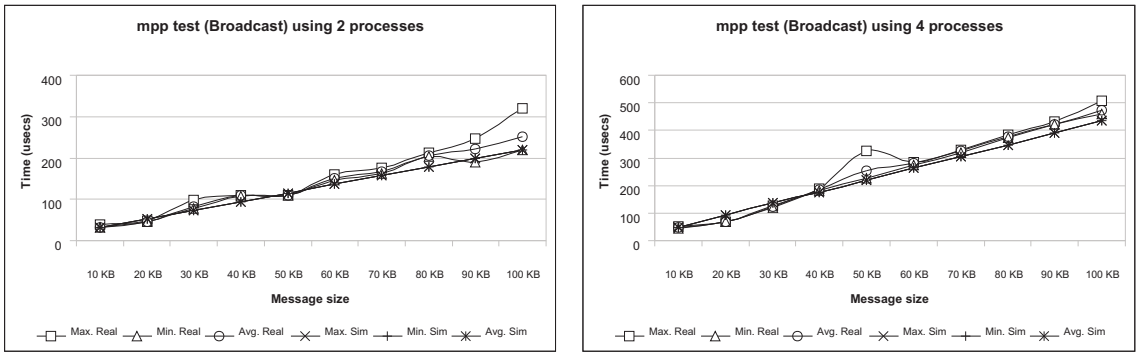


Figure 6.12: Execution of mppTest (Broadcast) in *environment_ 1* using 2 and 4 processes

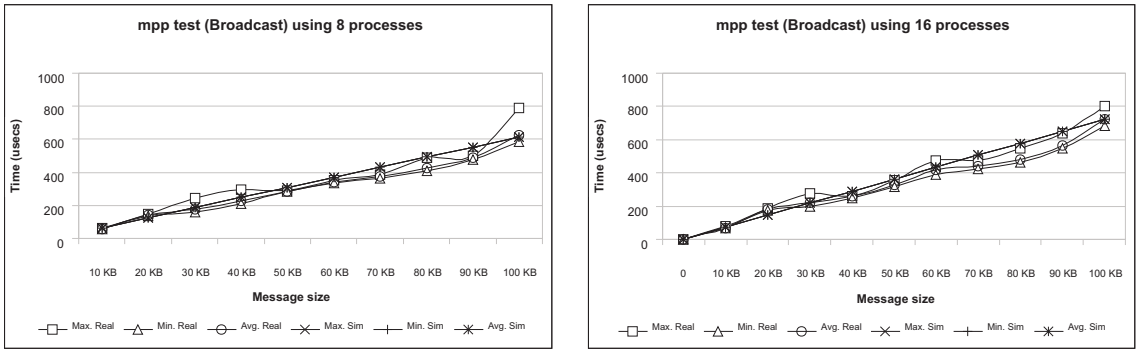


Figure 6.13: Execution of mppTest (Broadcast) in *environment_ 1* using 8 and 16 processes

(left chart), an error ratio of 4.9 % ± 0.4 is obtained. The Pearson's correlation coefficient is 0.9991. In those experiments that use message sizes from 100 KB to 1 MB (right chart) the error ratio obtained is 0.9 % ± 0.01, and the Pearson's correlation coefficient is 0.9999.

Figure 6.16 shows the results obtained in the execution of the Round-Trip benchmark in *environment_ 2* using 16 processes. When messages from 10 KB to 100 KB are used (left chart), an error ratio of 3.3 % ± 0.3 is obtained. The Pearson's correlation coefficient is 0.9995. In those experiments that use message sizes from 100 KB to 1 MB (right chart) the error ratio obtained is 0.9 % ± 0.01, and the Pearson's correlation coefficient is 0.9999.

Both the low error ratio and the Pearson's coefficient obtained in simulations, show that the behavior of Real Environment and the corresponding model is very similar. Those values (see figures 6.15 and 6.16) are very similar to those obtained in the same experiments executed in *environment_ 1* (see figures 6.10 and figure 6.11). This means that, even when different network configurations and different models of communication networks are used, simulations maintain the same accuracy and high level of correlation with the real system.

The charts of figures 6.15 and 6.16 show that in almost all cases, simulation results are placed between the maximum and the minimum results obtained in the execution of the Real Application, apart from for some peaks when message sizes between 50 KB and 90 KB are used. However, simulation experiments fit better for larger message sizes than for small message sizes. The Pearson's correlation coefficients of those experiments are very close to 1, which means a very strong correlation between the Real Environment and the

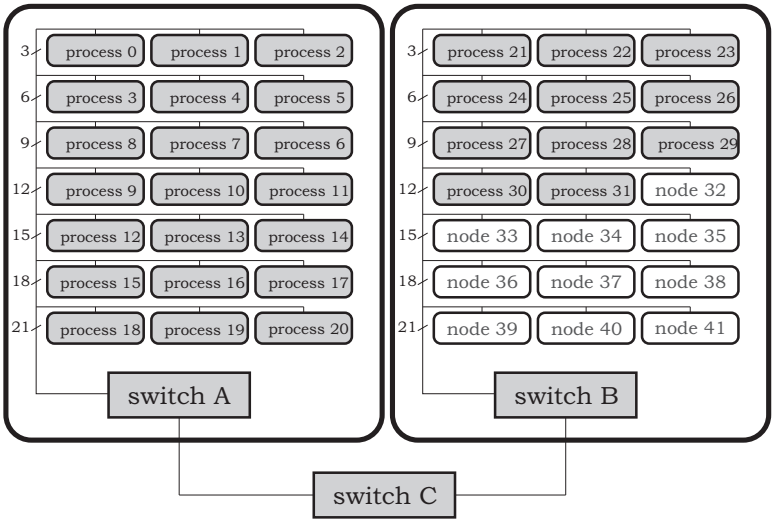


Figure 6.14: Configuration for executing mppTest in *environment_2*

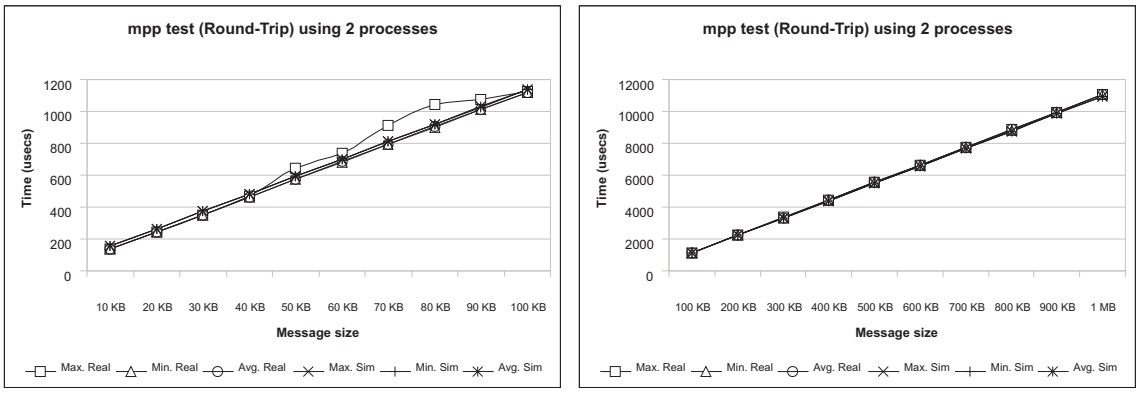


Figure 6.15: Execution of mppTest (Round-Trip) in *environment_2* using 2 processes

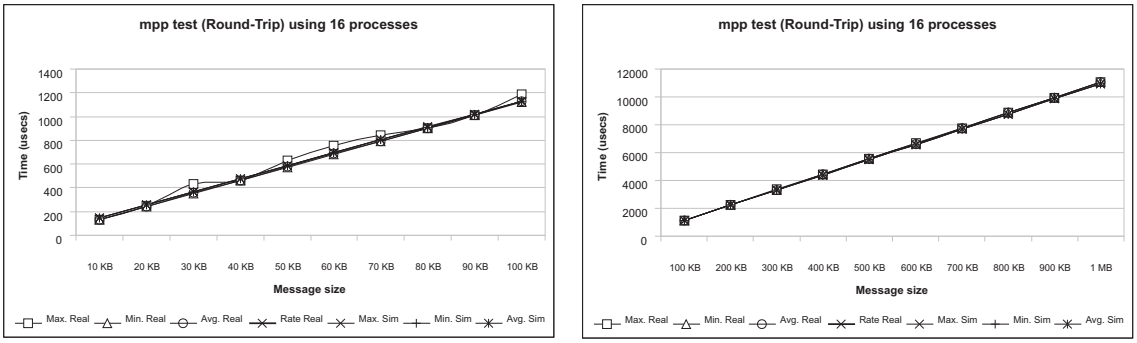


Figure 6.16: Execution of mppTest (Round-Trip) in *environment_2* using 16 processes

model.

Figures 6.17 and 6.18 show the results obtained in the execution of the Broadcast benchmark in *environment_2* using 2, 4, 8, and 16 processes. Message sizes used go from

10 KB to 100 KB. The error ratio obtained in those experiments is 4.2 % ± 1.3, 10.6 % ± 1.2, 8.6 % ± 1.6 and 8.8 % ± 1.4 for 2, 4, 8, and 16 processes respectively. The Pearson's correlation coefficient is 0.9952, 0.9947, 0.9673, and 0.9569 for 2, 4, 8, and 16 processes respectively.

The charts of figures 6.17 and 6.18 show that for a concrete range of processes and message sizes, the experiments executed in Real Environments produce some peaks in the overall performance. However, simulation results show an almost linear behavior, which hide those peaks. In those experiments, simulation results are very close to results obtained in Real Environments.

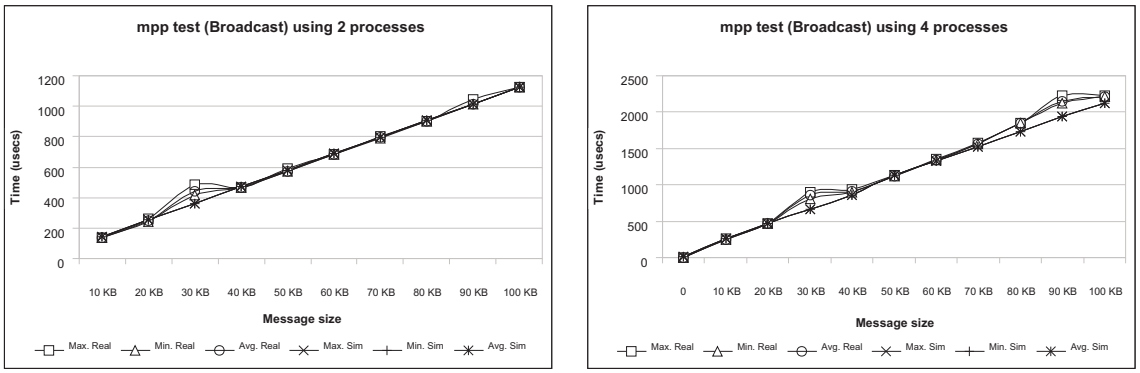


Figure 6.17: Execution of mppTest (Broadcast) in *environment_2* using 2 and 4 processes

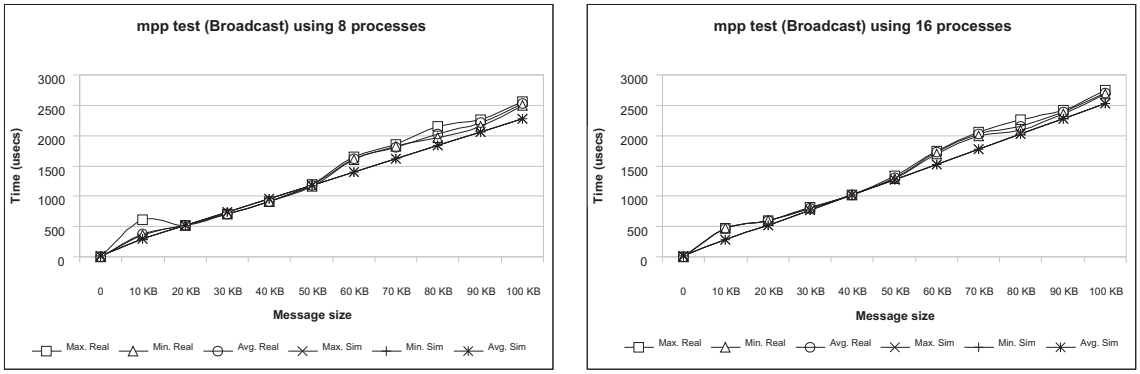


Figure 6.18: Execution of mppTest (Broadcast) in *environment_2* using 8 and 16 processes

In summary, Round-Trip experiments executed in *environment_2* have a slowdown factor between 4 and 5 approximately compared with the same experiments executed in *environment_1*. In the case of the Broadcast experiments there is a slowdown factor between 3.5 and 4.5 approximately between experiments executed in *environment_2* and experiments executed in *environment_1*. Those slowdown factors are reflected equally in the analogous models. Thus, simulations show the same tendency and overall system performance than the corresponding analogous real systems. This difference of performance is caused mainly due to the network used in *environment_2* is much slower than the network used in *environment_1*, which affects to the overall experiment performance.

In general, variability in Real Environments is greater than the analogous models. This

is caused mainly by additional elements, specifically by software elements like operating system latencies and external noise, which are not modeled in the SIMCAN simulation platform. However, this simulation platform is targeted to calculate the overall system performance, instead of the variability produced in a concrete system. The simulation results fit well with those obtained in the Real Environment, achieving low error ratios between the real system and the model.

The simulation of Round-Trip experiments achieves more accurate results than broadCast experiments. It can be caused due to the implementation of the broadCast call in MPICH distribution is more optimized than the implementation of the same call provided by SIMCAN. Besides this difference, the Pearson's coefficient shows that all tendencies of the experiments executed in Real Environments, are the same that those obtained in the analogous models. Moreover, this tendency is maintained even when both architectural and configuration changes are applied in those environments where experiments are executed.

6.4.3 BIPS3D application experiments

Once the storage and network systems have been analyzed, a parallel application that uses both systems will be tested, in order to check the accuracy of a model that uses both systems. The application used for this purpose is BIPS3D. In this section are explained the results achieved of executing BIPS3D in two environments using different configurations.

First configuration follows a schema where only one process, called master process, is in charge of performing I/O operations. This process reads the initial mesh from the storage node (using NFS), splits the mesh in sub-domains and then sends each sub-domain to the corresponding process. This mesh is stored in a storage node (labeled with "**Data (NFS)**") which exports a Network File System to computing nodes. The rest of processes, called slave processes, are executed on computing nodes and receive this data from the master process. Then, slave processes perform the corresponding operations and send the results back to the master process. Finally, master process writes the corresponding results in the storage node (labeled with "**Data (NFS)**") using NFS.

Second configuration follows a schema where only one process is in charge of reading the initial data, but all processes write results using parallel I/O. Initially, master process reads the initial mesh from the storage node (using NFS), splits the mesh in sub-domains and then sends each sub-domain to the corresponding process. This mesh is stored in a storage node (labeled with "**Data (NFS)**") which exports a Network File System to computing nodes. The rest of processes, called slave processes, are executed on computing nodes and receive this data from the master process. Then, slave processes perform the corresponding operations and write the corresponding results using a parallel file system (PVFS). This data is then stored in the storage nodes labeled with "**PVFS server X**". Depending on the configuration chosen, 2 or 4 PVFS servers are used.

In the charts showed in this section, the times obtained in the execution of each experiment have been grouped in three categories: time spent by the storage system, time spent by the network system, and total time needed for executing the experiment completely.

6.4.3.1 Experiments using sequential I/O with NFS servers

Figure 6.19 shows the configuration for the execution of BIPS3D in *environment_1* using a NFS server.

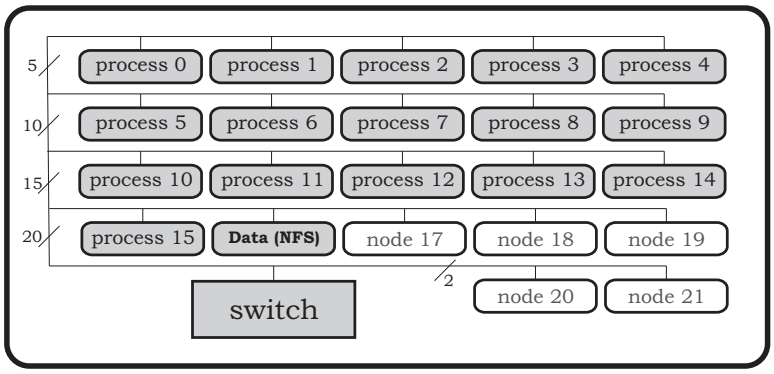


Figure 6.19: Configuration for executing BIPS3D in *environment_1* using a NFS server

Figure 6.20 shows the execution times of BIPS3D using 2, 4, 8, and 16 processes in *environment_1*. All I/O operations in those experiments are performed through NFS using one server. Left chart shows the times of the master process (process 0). Right chart shows the average times of slave processes. The Pearson's correlation coefficient is 0.9991 for the master process and 0.999 for the slave processes. The error ratio obtained in the simulation of the master process is 1.3 % ± 1.0 in the best case, and 4.1 % ± 0.2 in the worst case. Similarly, the error ratio obtained in the simulation of the slave processes is 2.7 % ± 0.2 in the best case, and 12.1 % ± 0.2 in the worst case.

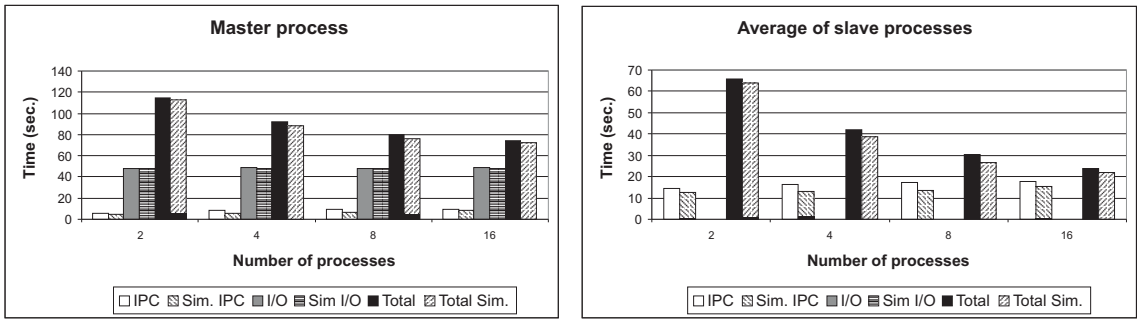


Figure 6.20: Execution of BIPS3D in *environment_1* using a NFS server

Due to master process performs all I/O operations; a linear increasing of performance can be appreciated when number of processes increases as well, but the I/O performance remains constant. This is caused due to the level of parallelism increases and then CPU calculations are distributed among more nodes, decreasing the total execution time. This tendency is also obtained in the model. The best accuracy is obtained in the simulation of the master process, which obtain lower error ratios. The slave processes are more difficult to predict accurately. However, the Pearson's coefficients are very strong in all cases.

Figure 6.21 shows the configuration for the execution of BIPS3D in *environment_2* using a NFS server.

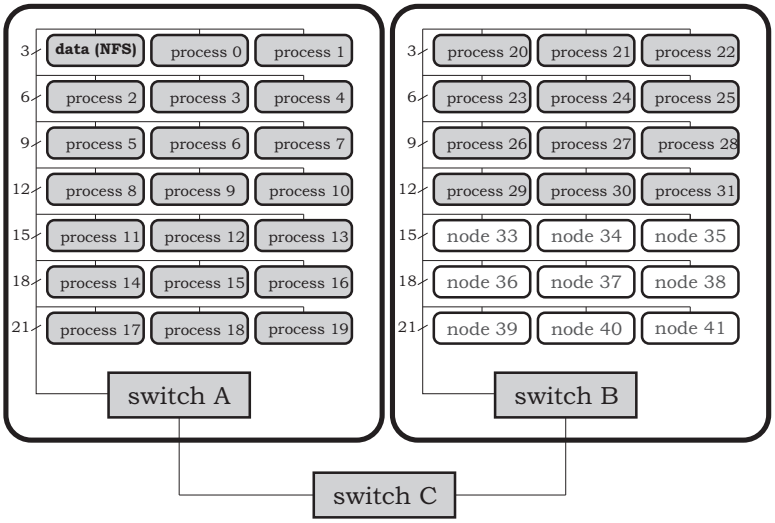


Figure 6.21: Configuration for executing BIPS3D in *environment_2* using a NFS server

Figure 6.22 shows execution times of BIPS3D using 2, 4, 8, 16 and 32 processes in *environment_2*. All I/O operations in this experiment are performed through NFS using one server. Left chart shows the times of the master process (process 0) and right chart shows the average times of all slave processes. The Pearson's correlation coefficient is 0.9995 for the master process and 0.9993 for the slave processes. The error ratio obtained in the simulation of the master process is 0.7 % ± 0.1 in the best case, and 1.6 % ± 0.3 in the worst case. Similarly, the error ratio obtained in the simulation of the slave processes is 6.5 % ± 0.1 in the best case, and 15.5 % ± 0.03 in the worst case.

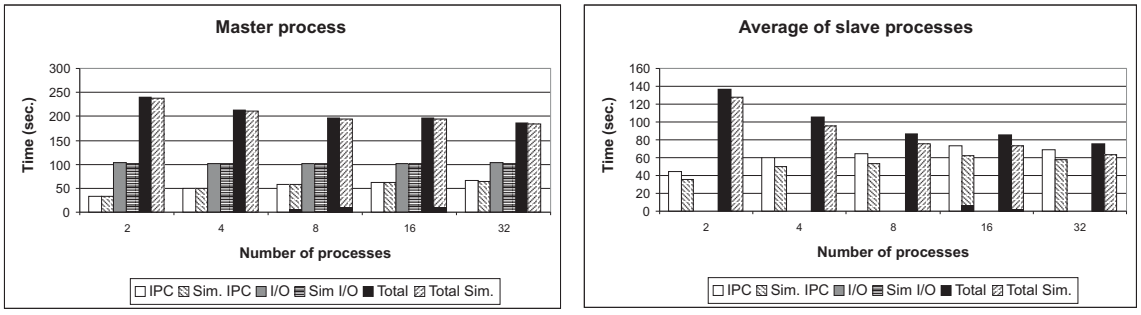


Figure 6.22: Execution of BIPS3D in *environment_2* using a NFS server

Both the error ratio and the Pearson's coefficient obtained in experiments executed in *environment_2* are very similar to those obtained in the same experiments executed in *environment_1*, which reaffirm the accuracy of the simulation platform when changes in the hardware architecture are made.

The charts that show the results obtained in the execution in those experiments in both *environment_1* and *environment_2* (see figures 6.20 and 6.22), show a linear increasing of performance when the number of processes increases. Both I/O and IPC times obtained in *environment_2* are greater than those obtained in *environment_1*. This difference of performance is caused due to the network used in *environment_2* is much slower than

the network used in *environment_1*. Those tendencies are also reflected in the simulation results.

6.4.3.2 Experiments using parallel I/O with NFS and PVFS servers

Figure 6.23 shows the configuration for the execution of BIPS3D in *environment_1* using both NFS and 2-4 PVFS servers.

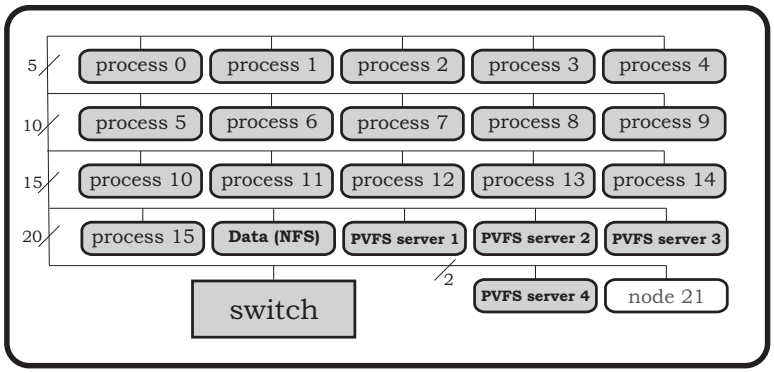


Figure 6.23: Configuration for executing BIPS3D in *environment_1* using a NFS and PVFS servers

Figure 6.24 shows the execution times of BIPS3D in *environment_1* using 1 NFS server and 2 PVFS servers. The Pearson's correlation coefficient is 0.9998 for the master process and 0.9994 for the slave processes. The error ratio obtained in the simulation of the master process is 5.3 % ± 0.9 in the best case, and 8.7 % ± 0.7 in the worst case. Similarly, the error ratio obtained in the simulation of the slave processes is 1.4 % ± 0.4 in the best case, and 9.0 % ± 0.5 in the worst case.

In those experiments, it is important to mention that the level of accuracy obtained in the storage system, when the parallel file system PVFS is used. This level of accuracy is obtained due to the use of the generic model of parallel file systems provided by SIMCAN, which has been adapted with the same the parameters used for configuring PVFS.

The performance obtained in the experiments that use parallel I/O is greater than performance obtained in those experiments that use sequential I/O. This is caused due to result files are written in parallel. Thus, I/O times are reduced considerably compared to sequential I/O (see figure 6.20). Moreover, IPC times obtained in slave processes decrease, because the use of parallel I/O reduces considerably the number of communications compared with the schema that uses only NFS servers (see section 6.4.3.1). In general, simulation times are close to real execution achieving the same behavior.

Figure 6.25 shows the execution times of BIPS3D in *environment_1* using 1 NFS server and 4 PVFS servers. The Pearson's correlation coefficient is 0.9998 for the master process and 0.9998 for the slave processes. The error ratio obtained in the simulation of the master process is 2.9 % ± 0.8 in the best case, and 11.2 % ± 0.5 in the worst case. Similarly, the error ratio obtained in the simulation of the slave processes is 1.5 % ± 0.4 in the best case, and 11.2 % ± 0.4 in the worst case. Those charts show that using 4 servers does not assure an increasing of performance. Thus, compared with the experiments that use 2 PVFS servers (see figure 6.24), the results obtained are just slightly better. This is

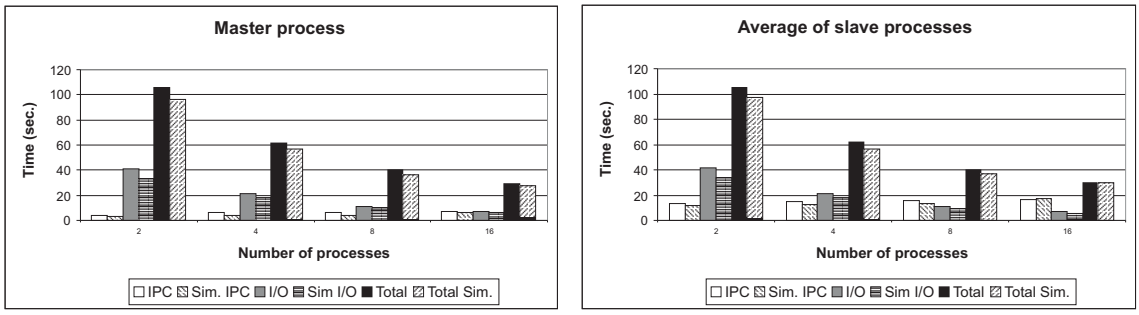


Figure 6.24: Execution of BIPS3D in *environment_1* using a NFS server and 2 PVFS servers

caused due to the level of parallelism can't be increased although the number of servers increases. However, those conclusions can be obtained with the simulation results because simulation has the same tendency than the execution in the Real Environment.

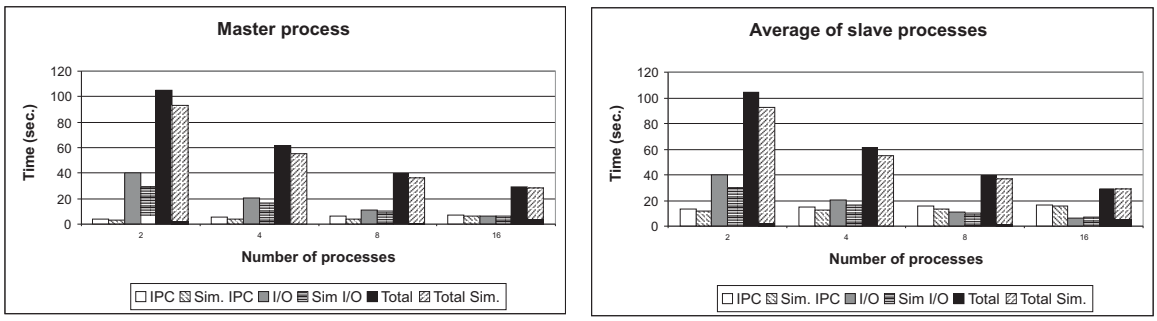


Figure 6.25: Execution of BIPS3D in *environment_1* using a NFS server and 4 PVFS servers

Figure 6.26 shows the configuration for the execution of BIPS3D in *environment_2* using 1 NFS server and 2 PVFS servers. Finally, figure 6.27 shows the configuration for executing BIPS3D in *environment_2* using both a NFS server and 4 PVFS servers.

Figure 6.28 shows the execution times of BIPS3D in *environment_2* using 1 NFS server and 2 PVFS servers. The Pearson's correlation coefficient is 0.9992 for the master process and 0.9985 for the slave processes. The error ratio obtained in the simulation of the master process is 1.1 % $\pm$ 0.2 in the best case, and 9.6 % $\pm$ 0.3 in the worst case. Similarly, the error ratio obtained in the simulation of the slave processes is 1.1 % $\pm$ 0.1 in the best case, and 8.6 % $\pm$ 0.8 in the worst case.

As occurs in the experiments executed in *environment_1* that use parallel I/O, the total execution time is considerably reduced compared with the same experiments that use sequential I/O. Simulation times fit very well with execution times obtained in Real Environments for all experiments. Also, IPC times decreases because the number of communications decreases as well using this schema.

Figure 6.29 shows the execution times of BIPS3D in *environment_2* using 1 NFS server and 4 PVFS servers. The Pearson's correlation coefficient is 0.9991 for the master process and 0.999 for the slave processes. The error ratio obtained in the simulation of the master process is 1.9 % $\pm$ 0.4 in the best case, and 8.6 % $\pm$ 0.7 in the worst case.

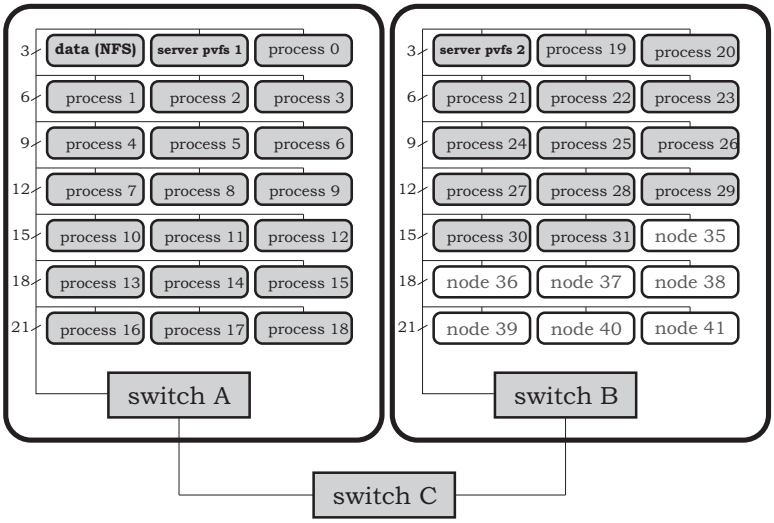


Figure 6.26: Configuration for executing BIPS3D in *environment_2* using a NFS and 2 PVFS servers

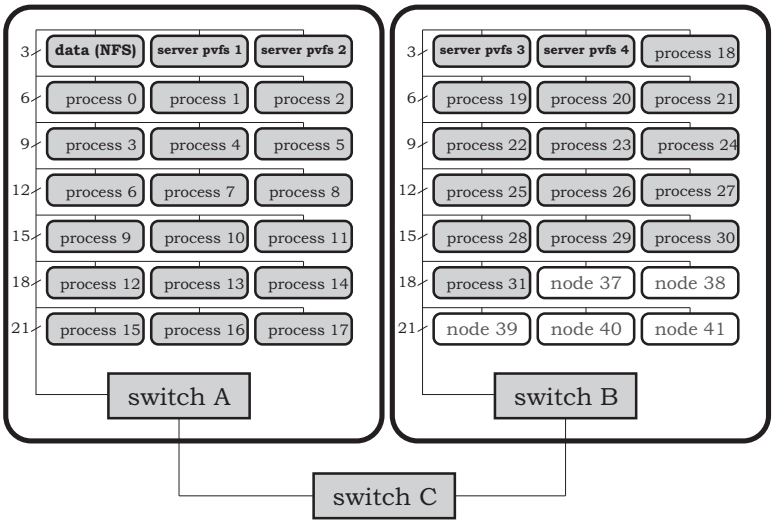


Figure 6.27: Configuration for executing BIPS3D in *environment_2* using a NFS and 4 PVFS servers

Similarly, the error ratio obtained in the simulation of the slave processes is 1.5 % ± 0.3 in the best case, and 8.4 % ± 0.1 in the worst case. Those charts show that using 4 PVFS servers instead of 2 provides just a slight increasing of performance, as occurs with the same experiments executed in *environment_1* (see figure 6.25).

In general, simulation results fit well with the execution in Real Systems. Moreover, Pearson's coefficients of those experiments are very close to 1, which means a very strong dependence between simulation and executions in Real Environments. Comparing the results shown in the charts of this section, we can see that results obtained in the simulations that use only one NFS server, fits well with results obtained in the analogous Real Environments. However, the error ratio increases slightly when parallel file systems are used.

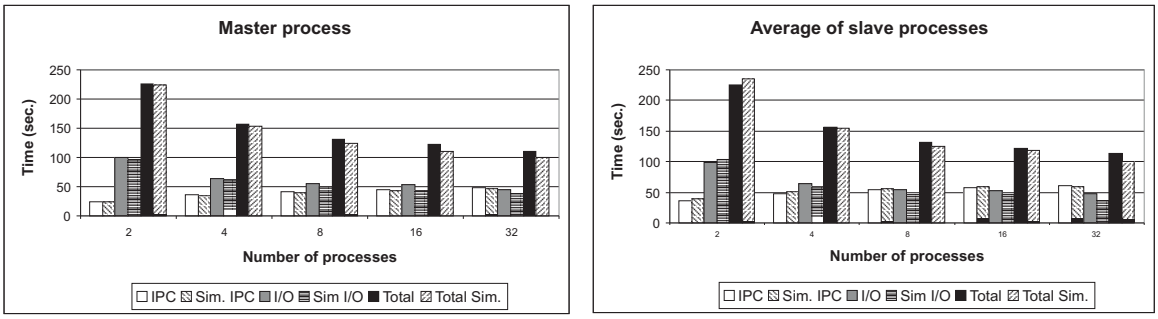


Figure 6.28: Execution of BIPS3D in *environment_2* using a NFS server and 2 PVFS servers

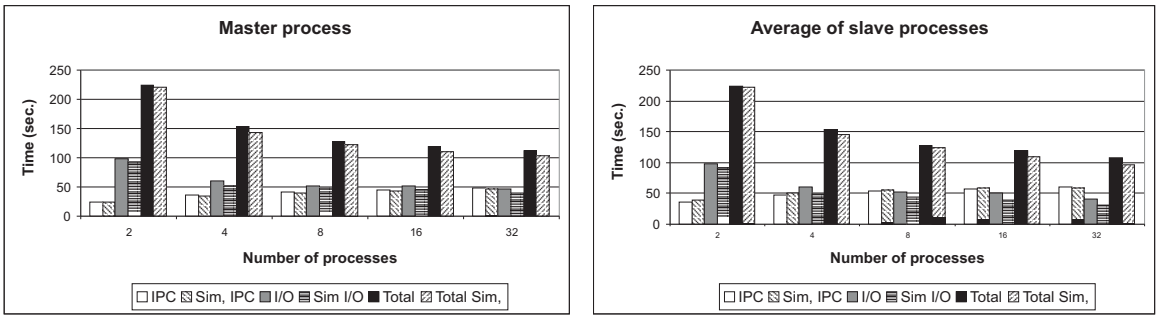


Figure 6.29: Execution of BIPS3D in *environment_2* using a NFS server and 4 PVFS servers

This is caused due to the different implementations of the PVFS and the generic model of parallel file systems provided by SIMCAN. This minimal discrepancy is unavoidable if we want that the implementation of the generic model provided by SIMCAN can be adjusted for modeling several kinds of parallel file systems. In the case of that a higher level of accuracy is needed, a new model that simulates the concrete parallel file system used would be completely implemented in the simulation platform.

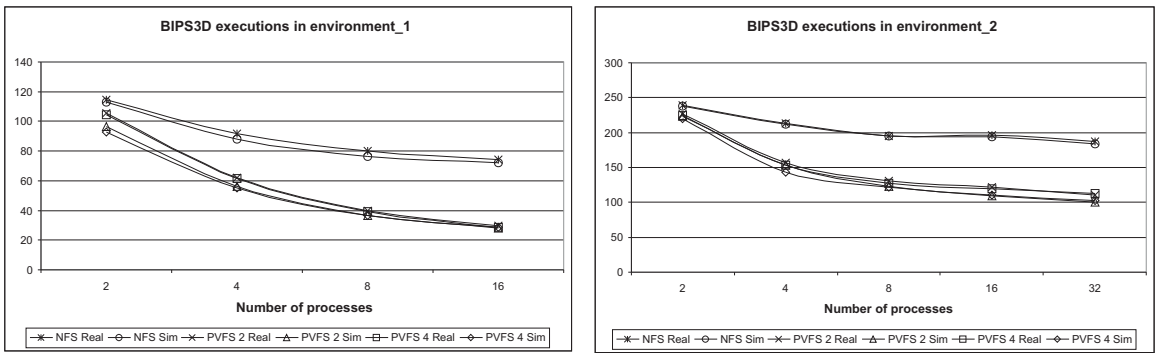


Figure 6.30: Comparative of executing BIPS3D using different I/O configurations

Figure 6.30 shows a comparative of the previous experiments of BIPS3D using several I/O configurations. Left chart shows the results of those experiments executed in *environment_1*. Right chart shows the results of those experiments executed in *environment_2*.

Both charts show the same behavior for executions in Real and Simulated Environments. When the number of processes increases, the performance obtained increases as well. This occurs for all experiments because the level of parallelism is also increased.

Sequential I/O is the configuration that provides the worst performance. Otherwise, experiments executed in *environment_1* show a more aggressive increasing of performance than experiments executed in *environment_2*. This behavior is caused due to the network used in *environment_1* is much faster, which causes a very important impact on the overall system performance.

Experiments that use parallel I/O obtain a great increasing of performance, which can be appreciated in figure 6.30. As occurs with sequential I/O, experiments executed in *environment_1* have a more aggressive increasing of performance that experiments executed in *environment_2*. However, increasing the number of PVFS servers up to 4 does not provide the same increasing of performance, obtaining almost the same results than using 2 PVFS servers.

In order to check the capacity of SIMCAN for facing changes in the hardware configurations, a comparison between same experiments executed in several environments using different configurations can be very useful. Then, a good measure for checking this feature can be the slowdown produced in the overall system performance, when the same experiment is executed in different scenarios. Then, in this section the slowdown provided by executing the same experiments in *environment_1* and *environment_2* is calculated.

Those experiments that use sequential I/O obtain a slowdown factor from 2,09 to 2,63 in the Real Environment, and from 2,12 to 2,71 in the Simulated Environment. When the parallel I/O is used, the slowdown factor obtained goes from 2,14 to 4,14 in the Real Environment, and from 2,41 to 4,20 in the Simulated Environment. When parallel I/O is used, a great increasing of performance can be appreciated compared to sequential I/O.

Those slowdown factors are caused because the network used in *environment_2* is not as fast as network used in *environment_1*. Moreover, simulations show the same tendency as the execution of the same experiments in the Real Environments. It is important to note that, although the absolute error value obtained in those experiments is greater than the absolute error value obtained in previous experiments, the relative results in both environments shows the same slowdown factors. Thus, the overall system performance estimated using SIMCAN is the same that performance calculated in the Real Environment.

6.4.4 Summary

This section shows a summary that contains the results obtained in all performed experiments of the sections 6.4.1, 6.4.2 and 6.4.3. Table 6.5 shows the statistical analysis performed using results obtained in the execution of the experiments in *environment_1*. Similarly, table 6.6 shows the statistical analysis performed using results obtained in the execution of the experiments in *environment_2*.

The meaning of each column in following described. First column (Experiment description) is a brief description of the experiment executed. Second column (# processes) is the number of processes involved in the experiment. Third column (Error) is the error ratio between the results obtained in the execution of the experiment in a Real Environment, and the results obtained the execution of the same experiments in a Simulated Environ-

ment (the closer to 0, the better). Fourth column (Pearson's coefficient) is the Pearson's correlation coefficient, which calculates how correlated are the results obtained in the Real Environment and the results obtained in the Simulated Environment (the closer to 1, the better). Finally, fifth column (Figure) shows the figure that contains the corresponding chart associated to each experiment.

Following list shows an overview of the main conclusions obtained by analyzing the results showed in tables 6.5 and 6.6.

- The most important conclusion obtained in this evaluation process is that, each model used shows the same tendency that the analogous Real Environment. This means that the estimated performance in the model fits with that obtained in the Real System, even when a small error ratio is obtained.
- In general, the error ratio obtained in all experiments is low.
- The variability achieved in the results is greater in those experiments executed in Real Environments than in those experiments executed in Simulated Environments. This is caused due to additional elements affect in the overall system performance. Those elements are mainly software elements, like latencies caused by the operating system, external noise in the communication network, etc. Due to those additional elements are not modeled in SIMCAN, the variability of the results obtained in the Simulated Environment is practically nonexistent.
- Due to the performance of the system analyzed can be estimated using simulated models, the variability existent in real systems does not have an important impact in the accuracy of the simulation platform. However, the main objective of this simulation platform is not to calculate the variability in a concrete system, but estimate the achieved performance in it.
- When the results obtained in the same experiments executed in different environment are compared with the analogous models, the slowdown factor obtained is very close. This means that the simulation platform provides a great level of accuracy when both configuration and architectural changes are made.
- In general, results obtained in simulation are slightly lower than results obtained in the analogous Real Environments. This difference of performance is caused due to external elements that causes variability in the results, are not modeled in the simulation platform. However, the error ratio obtained is low.
- In several cases where the software implementation can be achieved using several alternatives, like the MPI distribution used, or a concrete parallel file system, a higher difference of performance between the model and the Real Environment can be appreciated. The main cause of this is due to this simulation platform provides generic schemas for modeling a wide range of architectures, which entails a loss of precision when concrete software is used.
- In all cases, the Pearson's coefficient is very close to 1, which means a very strong correlation between the Real Environment and the model.

Experiment description	# processes	Error	Pearson's coefficent	Figure
IOZone local	1	3.36 % ± 0.52	N/A	figure 6.7
IOZone NFS	1	4.39 % ± 0.87	N/A	figure 6.7
mppTest Round-Trip (10KB)	2	7.2 % ± 0.6	0.9842	figure 6.10
mppTest Round-Trip (100KB)	2	10.2 % ± 0.9	0.9809	figure 6.10
mppTest Round-Trip (10KB)	16	7.7 % ± 0.6	0.9827	figure 6.11
mppTest Round-Trip (100KB)	16	8.9 % ± 0.8	0.9852	figure 6.11
mppTest Broadcast	2	9.9 % ± 1.1	0.9762	figure 6.12
	4	11.9 % ± 1.6	0.9905	figure 6.12
	8	11.3 % ± 0.9	0.9805	figure 6.13
	16	10.8 % ± 1.0	0.9799	figure 6.13
BIPS3D NFS	2 (master)	1.3 % ± 1.0	0.9991	figure 6.20
	2 (slaves)	2.7 % ± 0.2	0.999	figure 6.20
	4 (master)	3.6 % ± 0.1	0.9991	figure 6.20
	4 (slaves)	8.1 % ± 0.2	0.999	figure 6.20
	8 (master)	4.1 % ± 0.2	0.9991	figure 6.20
	8 (slaves)	12.1 % ± 0.2	0.999	figure 6.20
	16 (master)	2.9 % ± 0.9	0.9991	figure 6.20
	16 (slaves)	9.1 % ± 0.2	0.999	figure 6.20
BIPS3D PVFS (2 servers)	2 (master)	8.3 % ± 0.4	0.9998	figure 6.24
	2 (slaves)	8.0 % ± 0.3	0.9994	figure 6.24
	4 (master)	8.6 % ± 0.6	0.9998	figure 6.24
	4 (slaves)	9.0 % ± 0.5	0.9994	figure 6.24
	8 (master)	8.7 % ± 0.7	0.9998	figure 6.24
	8 (slaves)	8.1 % ± 0.5	0.9994	figure 6.24
	16 (master)	5.3 % ± 0.9	0.9998	figure 6.24
	16 (slaves)	1.4 % ± 0.4	0.9994	figure 6.24
BIPS3D PVFS (4 servers)	2 (master)	11.2 % ± 0.5	0.9998	figure 6.25
	2 (slaves)	11.2 % ± 0.4	0.9998	figure 6.25
	4 (master)	10.1 % ± 0.9	0.9998	figure 6.25
	4 (slaves)	9.9 % ± 0.8	0.9998	figure 6.25
	8 (master)	6.7 % ± 1.0	0.9998	figure 6.25
	8 (slaves)	6.7 % ± 0.7	0.9998	figure 6.25
	16 (master)	2.9 % ± 0.8	0.9998	figure 6.25
	16 (slaves)	1.5 % ± 0.4	0.9998	figure 6.25

Table 6.5: Statistical analysis overview of experiments executed in *environment_1*

Experiment description	# processes	Error	Pearson's coefficent	Figure
IOZone local	1	4.24 % ± 1.4	N/A	figure 6.8
IOZone NFS	1	7.38 ± 1.79	N/A	figure 6.8
mppTest Round-Trip (10KB)	2	4.9 % ± 0.4	0.9991	figure 6.15
mppTest Round-Trip (100KB)	2	0.9 % ± 0.01	0.9999	figure 6.15
mppTest Round-Trip (10KB)	16	3.3 % ± 0.3	0.9995	figure 6.16
mppTest Round-Trip (100KB)	16	0.9 % ± 0.01	0.9999	figure 6.16
mppTest Broadcast	2	4.2 % ± 1.3	0.9952	figure 6.17
	4	10.6 % ± 1.2	0.9947	figure 6.17
	8	8.6 % ± 1.6	0.9673	figure 6.18
	16	8.8 % ± 1.4	0.9569	figure 6.18
BIPS3D NFS	2 (master)	0.7 % ± 0.1	0.9995	figure 6.22
	2 (slaves)	6.5 % ± 0.1	0.9993	figure 6.22
	4 (master)	0.8 % ± 0.1	0.9995	figure 6.22
	4 (slaves)	9.9 % ± 0.3	0.9993	figure 6.22
	8 (master)	0.7 % ± 0.2	0.9995	figure 6.22
	8 (slaves)	13.1 % ± 0.9	0.9993	figure 6.22
	16 (master)	1.2 % ± 0.3	0.9995	figure 6.22
	16 (slaves)	14.0 % ± 0.8	0.9993	figure 6.22
	32 (master)	1.6 % ± 0.3	0.9995	figure 6.22
	32 (slaves)	15.5 % ± 0.03	0.9993	figure 6.22
BIPS3D PVFS (2 servers)	2 (master)	1.1 % ± 0.2	0.9992	figure 6.28
	2 (slaves)	4.1 % ± 0.2	0.9985	figure 6.28
	4 (master)	2.4 % ± 0.1	0.9992	figure 6.28
	4 (slaves)	1.1 % ± 0.1	0.9985	figure 6.28
	8 (master)	5.9 % ± 0.2	0.9992	figure 6.28
	8 (slaves)	5.1 % ± 0.1	0.9985	figure 6.28
	16 (master)	9.6 % ± 0.3	0.9992	figure 6.28
	16 (slaves)	2.7 % ± 0.1	0.9985	figure 6.28
	32 (master)	9.3 % ± 2.3	0.9992	figure 6.28
	32 (slaves)	8.6 % ± 0.8	0.9985	figure 6.28
BIPS3D PVFS (4 servers)	2 (master)	1.9 % ± 0.4	0.9991	figure 6.29
	2 (slaves)	1.5 % ± 0.3	0.999	figure 6.29
	4 (master)	6.5 % ± 0.2	0.9991	figure 6.29
	4 (slaves)	4.9 % ± 0.2	0.999	figure 6.29
	8 (master)	4.4 % ± 0.2	0.9991	figure 6.29
	8 (slaves)	3.5 % ± 0.1	0.999	figure 6.29
	16 (master)	7.6 % ± 0.4	0.9991	figure 6.29
	16 (slaves)	8.4 % ± 0.1	0.999	figure 6.29
	32 (master)	8.6 % ± 0.7	0.9991	figure 6.29
	32 (slaves)	7.1 % ± 0.2	0.999	figure 6.29

Table 6.6: Statistical analysis overview of experiments executed in *environment_2*

Chapter 7

Scalability and Performance experiments

In this chapter, a set of experiments has been designed for analyzing and optimizing the system architecture that provides the best performance to execute specific applications. Therefore, those experiments are focused on simulating HPC environments by increasing both the size of the problem and the size of the environment.

Moreover, those tests also intent to demonstrate that SIMCAN is able to model and simulate HPC systems, and executing those simulations in a reasonable time frame. Thus, in order to fulfill this objective, both the execution time and the amount of memory needed for executing those simulations will be measured.

In order to simulating those experiments, both several architectures and scalable application models have been modeled.

7.1 Introduction

Nowadays, large-scale systems are increasing their role due to the fast evolution on computer networks and communication technologies. Also, those systems grow more complex with each generation, becoming difficult and time-consuming task analyzing and predicting the behavior of those systems.

At present there is not a specific approach that increases the performance and scalability to any application. Due to this reason, it is necessary to estimate the impact of any design feature or any application on the overall system performance. Predicting the impact of even small changes on the performance of complex system is a very difficult and non-trivial job. Aspects like detecting system bottlenecks and calculating the scaling degree that some algorithms could obtain are expensive and time-consuming tasks when they are performed on large computing networks.

The most commonly applications used on those systems are scientific applications, which perform massive I/O operations and requires high amount of computing. Each application has a concrete behavior, where some architectures work well when a concrete kind of application is executed in that environment, but the same architecture can suffer a performance drop with another application. In most cases, the common factor that makes

a performance drop on those systems is the storage subsystem. This is the reason why designing the correct I/O architecture results of extreme importance.

A set of experiments are defined with the purpose of optimizing the architecture that provides the better performance. Thence, the main purpose of those experiments is evaluating and analyzing how evolves both scalability and bottlenecks existent on a typical HPC multi-core architecture using different configurations. Those experiments are focused on simulating HPC environments by increasing both the size of the problem and the size of the environment. By the size of the problem we mean the size of the data set that a corresponding application has to process completely to consider its execution done. Similarly, by the size of the environment we mean the number of computing nodes where the application will be executed. Generally, the greater the size of the environment, the greater the performance obtained, and the higher amount of resources needed for simulating such environments.

Simulation of HPC systems remains to be a challenge due to the high number of issues that hamper this task. Basically, the main issue for achieving simulations of HPC systems is two-fold. First, the enormous amount of time required for executing those simulations. Second, the large amounts of memory required for simulating the high number of elements that constitute the model. In most cases, those systems contain thousands of computing nodes, a set of storage nodes, communication networks, and communication switches, whereof the algorithms required for modeling and simulating all those elements require huge amounts of CPU power.

Moreover, performance experiments achieved in this chapter are focused on calculating the time needed for executing the corresponding simulations. Thence, the main objective of those experiments is to calculate both the amount of time and memory needed for executing a concrete simulation, depending of the size of the environment to be modeled, and the resources available for executing such simulation

7.2 Designing application models

In this section, the proposed models for simulating the behavior of two typical applications are described. In most cases, those kinds of applications are executed in parallel among a set of nodes using the available resources of a distributed environment. Thence, the main objective of this process is to obtain scalable and flexible models, which can be executed in several environments using different configurations and architectural designs. Following sections describe the process for modeling the behavior of a typical HPC application, and the behavior of a typical checkpointing application.

7.2.1 Modeling a typical HPC application

Generally, the typical behavior of HPC applications consists on splitting a problem in a set of domains, where the size of each one of those domains is smaller than the size of the whole problem. Then, those domains are spread among a set of processes, which calculate those domains in parallel. Usually, those processes are distributed among a set of CPU cores in a distributed environment. Depending of the configuration used, computing nodes contain a different number of CPU cores. Therefore, processing those domains in parallel

must increase the performance of the overall execution of the application.

The behavior of a typical HPC application has been modeled using the schema shown in figure 7.1, which is based on the map-reduce model proposed by Google [DG08]. This model uses an initial data set as the size of the problem. By size of the problem we mean the amount of data that have to be processed in order to accomplish the execution of the application completely. Then, once the data set has been completely processed, the application can be considered done. In this model there are two kinds of processes, where each one is in charge of achieving a concrete set of tasks: coordinator processes (C) and worker processes (W). Those processes are grouped in frames, whereof each frame contains one coordinator process and a set of worker processes.

The model starts by reading a set of domains from the storage nodes (1). Coordinator processes perform this task. Then, each coordinator process sends the corresponding domain to the worker processes that it is in charge (2). Once a worker process receives a domain, this process starts to compute it (3). When a worker process finishes the processing of a current domain, then the resulting data is sent to its coordinator process (4). Finally, when a coordinator process receives the results of a given domain, then those results are written to disks (5) and next domain are read by the coordinator process, starting again in the step (1). Therefore, coordinator processes deliver domains to the worker processes until the data set is processed completely.

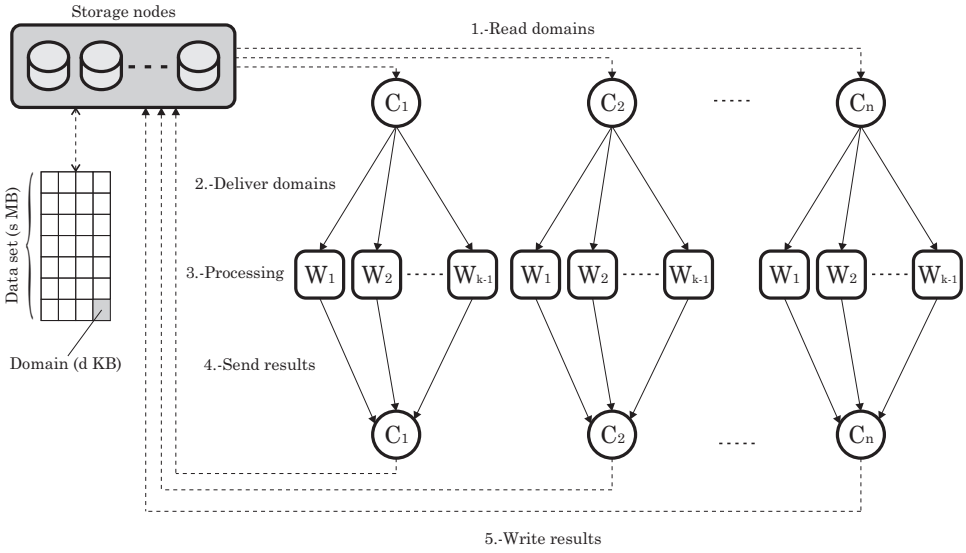


Figure 7.1: Proposed model of a typical HPC application behavior

The basic idea of this model is to customize the behavior of the application by using a set of parameters.

- Number of coordinator processes (n): Total number of coordinator processes.
- Size of processes frame (k): Size of each set of worker processes plus one coordinator process.
- Size of the initial data set (s): Size of the initial data set (in MB), which represents the size of the problem.

- Size of each domain (d): Size of each domain (in KB) which is delivered to worker processes.
- Size of partial results: Size of the resulting data (in KB) that is sent from the worker process to its corresponding coordinator process.
- Processing for each domain: Number of MIs (Million Instructions) needed for processing one domain.

The total number of processes involved in the application can be calculated using equation 7.1:

$$totalProcesses = k * n \tag{7.1}$$

Then, the number of domains will be given by the formula 7.2:

$$totalDomains = (s \cdot 1024)/d \tag{7.2}$$

Using those parameters, the behavior of this model can be customized depending of the objectives of the simulations. Moreover, this model is perfectly scalable because configuring the number of processes and the initial data size can set the scalability degree of the problem.

7.2.2 Modeling a typical checkpointing application

Application checkpointing is the act of saving the state of a computation such that, in the event of failure, it can be recovered with only minimal loss of computation. This is especially useful in areas such as computational biology where it is not unusual for an application to run for many weeks before to completion [WC06]. Otherwise, in distributed shared memory, checkpointing is a technique that helps tolerate the errors leading to lose the effect of work of long-running applications.

The checkpoint size is also of great concern when checkpointing large applications. Using standard user-level checkpointing techniques or kernel-level techniques typically save nearly the entire process state including data that need not be saved in order to recover the application. For example, in [BMP+04] the authors note that in "protein-folding applications on the IBM Blue Gene machine, an application-level checkpoint is a few megabytes in size whereas a full system-level checkpoint is a few terabytes."

The behavior of a typical checkpointing application has been modeled using the schema shown in figure 7.2. This schema shows a set of processes (P), where each process fulfills two phases. First phase consists on performing processing, and second phase consists on writing the state of the process to disk. The execution of those phases is called iteration.

In order to let users customize the behavior of the application, a set of parameters must be configured.

- Number of processes (n): Total number of processes.
- Processing for each domain: Number of MIs (Million Instructions) processed in first phase for each process.

- Size of process state: Size of process state (in KB) that is written to disk in each iteration for each process.
- Number of iterations: Number of iterations to be achieved for each process.

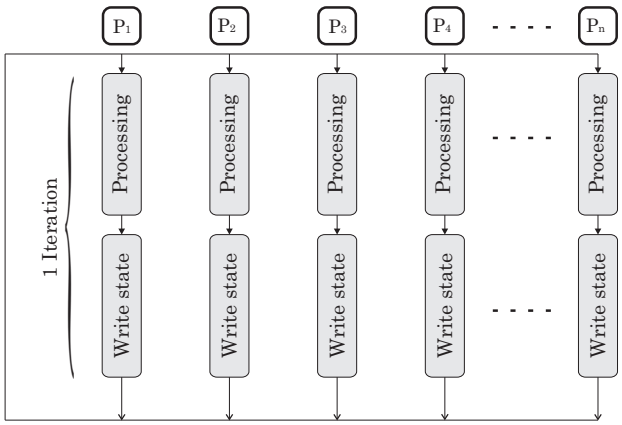


Figure 7.2: Proposed model of a typical checkpointing application behavior

7.3 Modeling a typical HPC multi-core architecture

Application-architecture combination has widespread applicability in distributed systems research. Thus, finding an architectural configuration for a concrete application is a key factor to obtain optimal performance.

Therefore, in order to analyze the execution of the application models described in section 7.2, a typical HPC environment has been modeled for this purpose. The main objective of this environment is to simulate application models using different configurations, such as the number of computing nodes, the number of CPU cores per computing node (multi-core architectures) and the number of storage nodes.

Figure 7.3 shows the basic schema of the proposed HPC environment model. Basically, this model consists of 6 elements, which are described below:

- Rack: This element is used in order to group computing nodes in large-scale systems. Due to managing large amounts of nodes hampers the task of configuring the entire system, racks contains a set of board nodes (b) in order to ease deploying tasks.
- Board node: This element is used with the same purpose as racks, for grouping and managing sets of nodes. This element contains a set of computing nodes (n) and one switch. That switch is used for interconnecting the set of nodes inside a node board with the rest of the architecture.
- Node: This element is used to perform processing. Usually those nodes contain a set of CPU cores for executing several processes in parallel.
- Storage node: Those elements are used for managing data.

- Switches: Those elements are used for interconnecting the elements of the HPC architecture.
- Communication network: This element defines the speed of each communication link in the architecture. Basically two parameters define this element: network bandwidth (measured in Mbps) and latency (measured in μs).

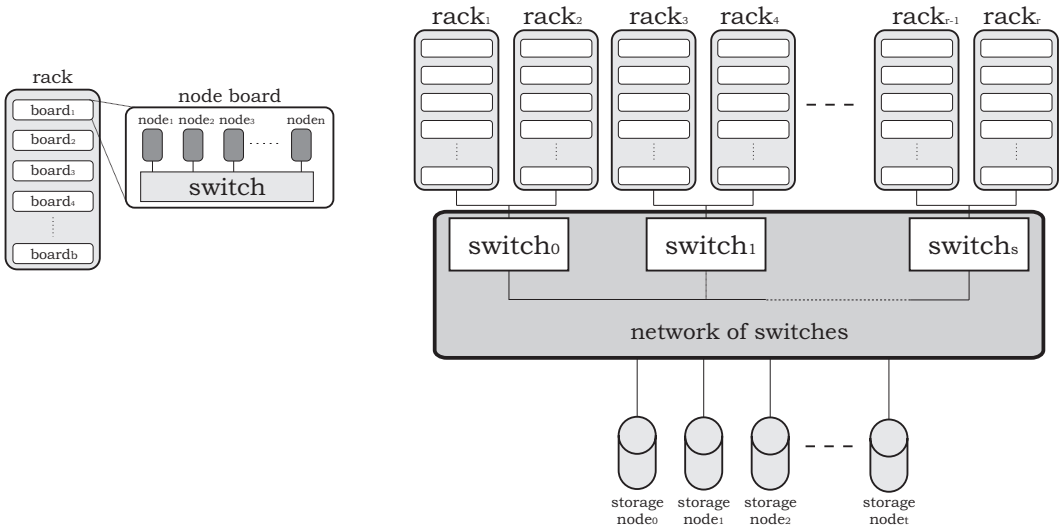


Figure 7.3: Proposed model of a typical HPC environment

7.4 Simulating application models in different architectures

In order to simulate the previous application models, three different environments have been modeled. Thus, each application model is executed in each architecture with the purpose of optimizing the best architectural configuration that provides the best performance.

First, a scenario that contains a total of 128 computing nodes have been modeled using the schema described in section 7.3, which is shown in figure 7.4.

Second, using the same schema defined in section 7.3, a scenario that contains a total of 1024 computing nodes has been modeled (see figure 7.7).

Finally, a many-core architecture has been used for simulating application models previously described. This architecture consists of a single node with a customizable number of CPU cores up to 512. This node is attached directly to a switch, which has also attached the storage system composed by a set of I/O servers. The schema of this architecture is shown is figure 7.11.

7.4.1 Simulating the HPC application model

This section presents the simulation experiments of the HPC application model using the three environments previously described (see figures 7.4, 7.7, and 7.11). In those experiments, the size of data set calculated by the application model is 256 MB. Thus, the main

Environment model configuration	Application model configuration
Racks: 2	Data-set: 256 GB
Boards per rack: 8	Domain size: 5 MB
Computing nodes per board: 8	Results size: 512 KB
Total number of computing nodes: 128	Size of processes frame: 32
CPU cores per node: 1, 2, 4, 8 and 16	Coordinator processes: 4, 8, 16, 32 and 64
CPU core speed: 20000 MIPS	Processing per domain: 500 MIs
Storage nodes: 1, 2, 4, 8 and 16	
File system: Parallel file system	
Network: Ethernet 1 Gbps / 10 Gbps	

Table 7.1: Environment configuration of HPC model experiments using 128 nodes

purpose of those experiments is to optimize the architectural configuration that provides the best performance. Basically, this process is achieved by modifying the architectural parameters, like the number of CPU cores per node, the number of I/O servers and the characteristics of the communication network.

Initially, the HPC application model has been simulated using a scenario that consists of 128 computing nodes (see figure 7.4). The configuration used for executing those experiments is shown in table 7.1, and the results obtained from those simulations are shown in figure 7.5. Those results are grouped in two charts, depending of the communication network used. Thus, experiments executed using a network of 1 Gbps correspond with left chart (see figure 7.5(a)), and experiments that use a network of 10 Gbps correspond with right chart (see figure 7.5(b)).

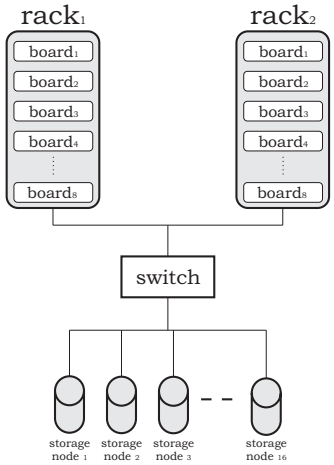
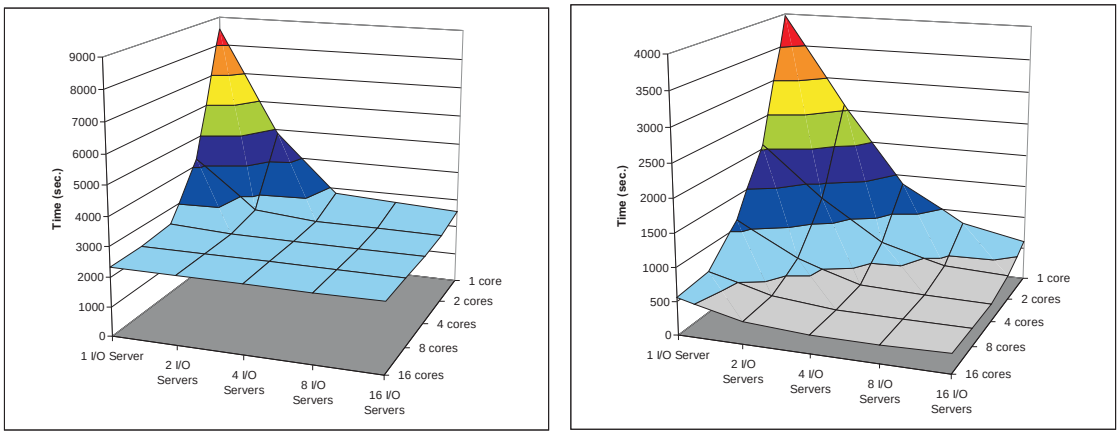


Figure 7.4: HPC scenario with 128 computing nodes

Those experiments show that there is a considerable system bottleneck when single-core CPUs and only 1 I/O server are used. This occurs because the level of parallelism in the system using this configuration is practically nonexistent. Otherwise, when the number of CPU cores per node or I/O servers increases, the overall system performance reflects a significant improvement.

Increasing the number of I/O servers produces a notable improvement in the overall system performance. This improvement is caused because the I/O operations of several



(a) Network of 1 Gbps (b) Network of 10 Gbps

Figure 7.5: Simulation of HPC model using 128 nodes

processes are executed in parallel when several I/O servers are used. Similarly, when the number of CPU cores per node increases, also increases the system performance. However, this improvement is not reflected in all cases. For instance, in the scenario that uses a network of 1 Gbps, using more than 2 I/O servers and more than 2 CPU cores per node does not provide a significant increasing of performance. Similarly, using a faster network produces a similar effect, but in this case the overall system performance remains stuck from 4 I/O servers and 4 CPU cores per node.

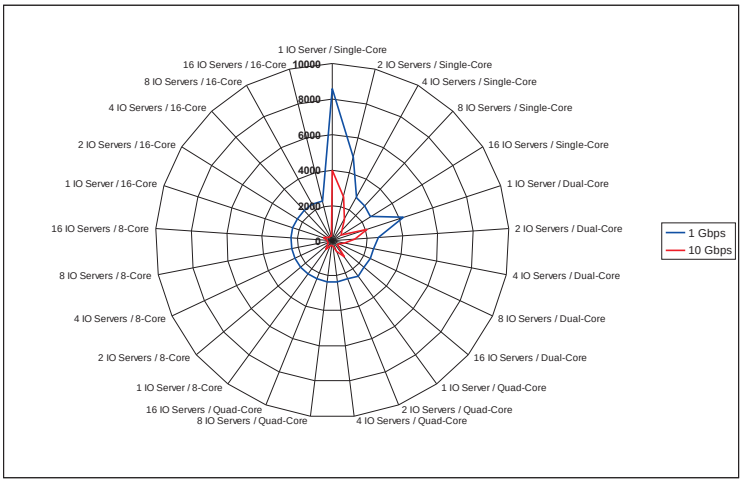


Figure 7.6: Simulation time of HPC experiments using 128 nodes and different networks

In conclusion, using a fast network in those experiments causes a great increasing of performance for all configurations (see figure 7.6). This occurs because the network acts as a system bottleneck, and increasing the network bandwidth the overall system performance increases as well. Otherwise, increasing the number of CPU cores and I/O servers does not warranty an increasing of performance because the application raises the maximum

Environment model configuration	Application model configuration
Racks: 8	Data-set: 256 GB
Boards per rack: 8	Domain size: 5 MB
Computing nodes per board: 16	Results size: 512 KB
Total number of computing nodes: 1024	Size of processes frame: 32
CPU cores per node: 1, 2, 4, 8 and 16	Coordinator processes: 4, 8, 16, 32 and 64
CPU core speed: 20000 MIPS	Processing per domain: 500 MIs
Storage nodes: 1, 2, 4, 8 and 16	
File system: Parallel file system	
Network: Ethernet 1 Gbps / 10 Gbps	

Table 7.2: Environment configuration of HPC model experiments using 1024 nodes

throughput provided by the system.

Next experiments are targeted to analyze the impact on the overall system performance when the number of computing nodes increases. Previous experiments were executed in a scenario that consists of 128 nodes using different CPU cores per node (from 1 to 16). In this case, the same application model (HPC model) is executed in a scenario that consists of 1024 computing nodes (see figure 7.7) using the configuration shown in table 7.2.

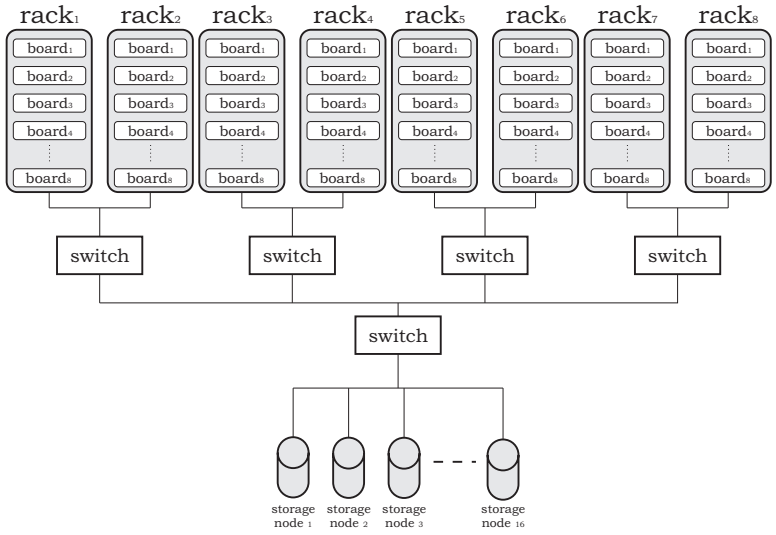
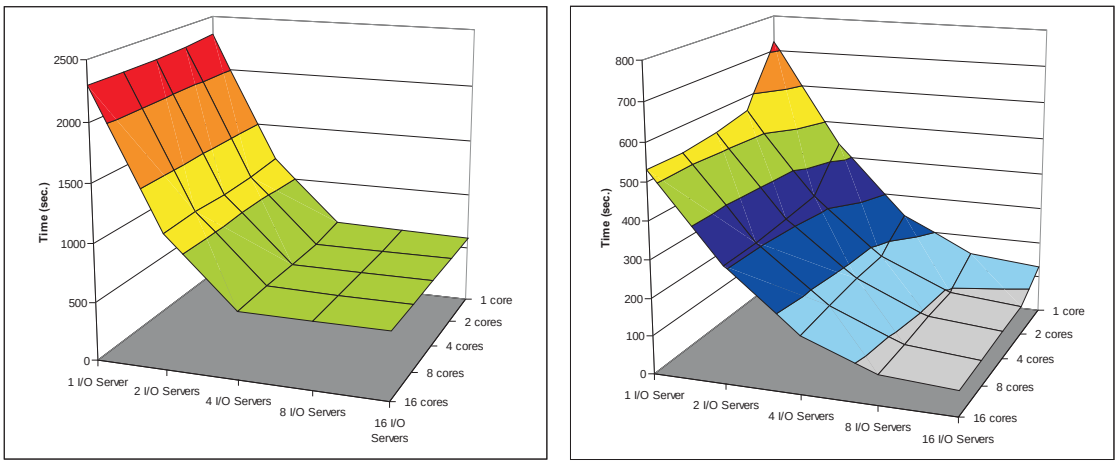


Figure 7.7: HPC scenario with 1024 computing nodes

Figures 7.8 and 7.9 show the results obtained from those simulations. Initially, figure 7.8(a) shows the results of the experiments using a network of 1 Gbps. Next, figure 7.8(b) shows the results of the experiments using two different networks, a 1 Gbps network that interconnects the racks with the network of switches (network for communications) and another network of 10 Gbps that interconnects the network of switches with the storage servers (I/O system network), Finally, figure 7.9 shows the results of the experiment using a network of 10 Gbps.

Those charts show clearly that using a different number of CPU cores per node in this scenario doesn't have a significant improvement on the overall system performance. Only when a network of 10 Gbps is used in the I/O system, dual-core CPUs provide a light

increasing of performance. In the rest of configurations, the overall system performance remains fixed.



(a) Network of 1 Gbps

(b) Mixed network of 1/10 Gbps

Figure 7.8: Simulation of HPC model using 1024 nodes

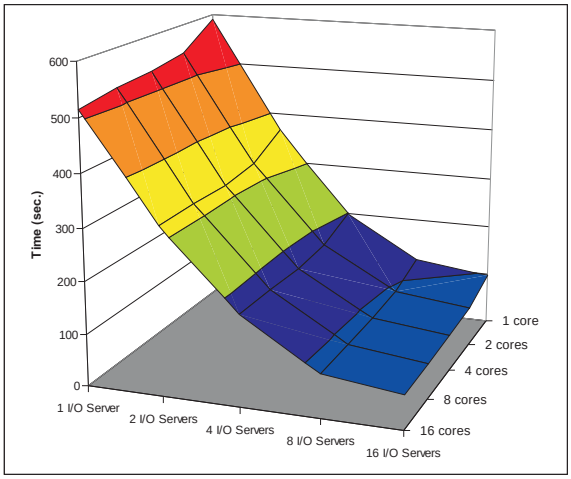


Figure 7.9: Simulation of HPC model using 1024 nodes and 10 Gbps network

Otherwise, increasing the number of I/O servers produces a significantly increasing of performance. This occurs because using 1024 computing nodes, the bottleneck of the system lays more aggressively in both the network and storage systems. Therefore, increasing the number of I/O servers, the level of parallelism in the storage system increases as well, alleviating the system bottleneck. In the first scenario (see chart 7.8(a)) the overall system performance is stuck from 4 I/O servers. Otherwise, the other scenarios (see charts 7.8(b) and 7.9) present an increasing of performance up to 8 I/O servers.

Comparing the experiments that use a mixed network (1 Gbps for communications and 10 Gbps for the storage subsystem) with those experiments that use a slow network (1

Gbps), we can conclude that the major part of the bottleneck lies in the storage system, because using the same bandwidth for communications and a faster network for the I/O subsystem, the overall system performance is clearly improved (see figure 7.8).

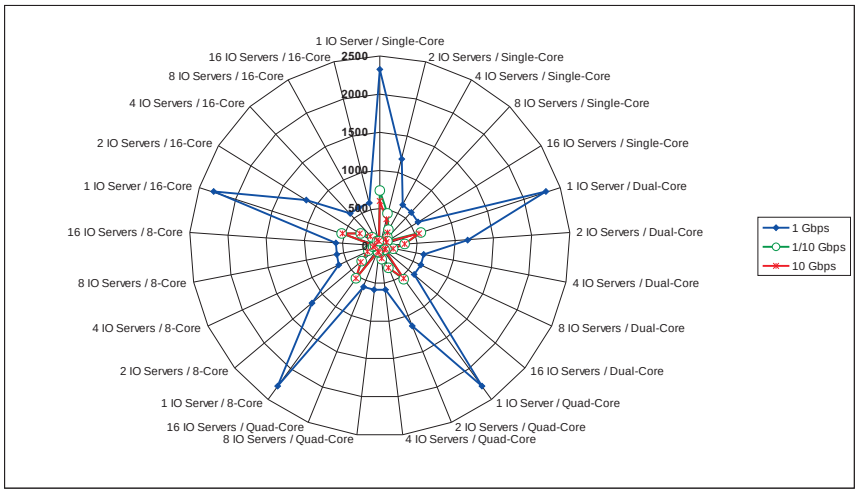


Figure 7.10: Simulation time of HPC experiments using 1024 nodes and different networks

In conclusion, the results of those experiments show that the I/O system is a critical factor in HPC systems, obtaining a greater performance by increasing the network bandwidth instead the number of CPUs per node or the number of I/O servers (see figure 7.10). When a 10 Gbps network is used in the I/O system, the overall system performance reflects a great improvement of performance, which is not such remarkable when a 1 Gbps network is used (see figure 7.9).

Finally, the HPC model experiments are executed in a many-core node. This node is configured for using different number of CPU cores, which goes from 2 to 512 (see figure 7.11). Table 7.3 shows the configuration for those experiments.

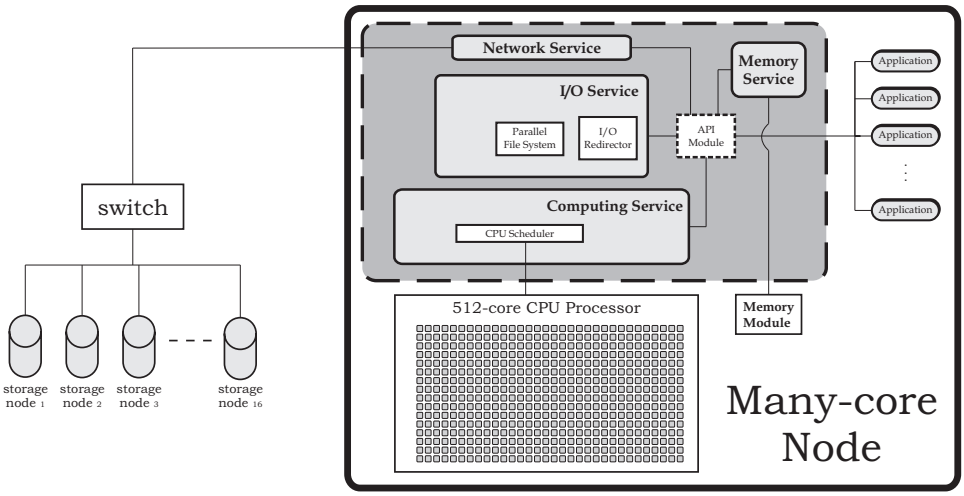
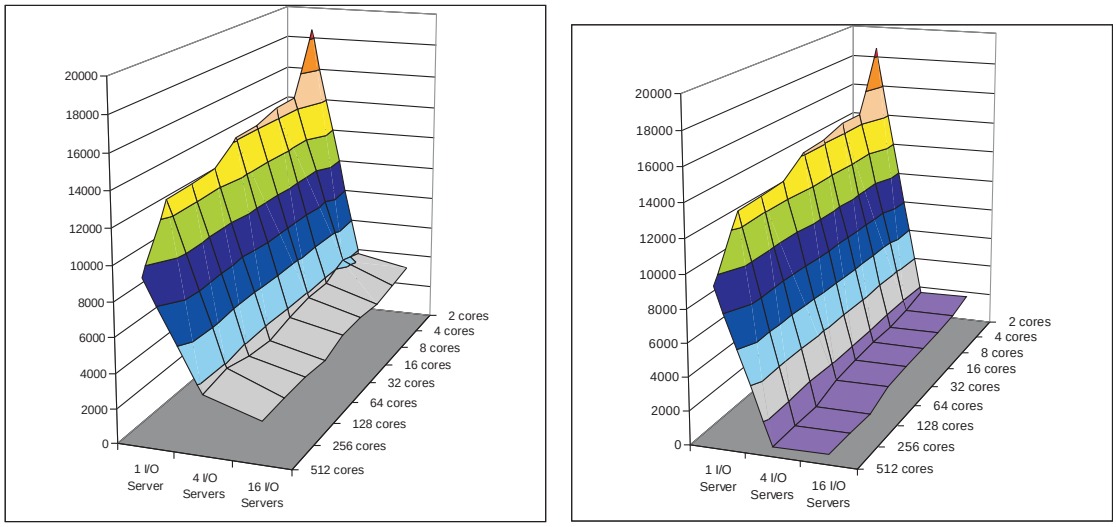


Figure 7.11: Many-core node scenario

Environment model configuration	Application model configuration
Computing nodes: 1	Data-set: 256 GB
CPU cores per node: 2, 4, 8, 16, 32, 64, 128, 265 and 512	Domain size: 5 MB
CPU core speed: 20000 MIPS	Results size: 512 KB
Storage nodes: 1, 4 and 16	Size of processes frame: 2, 4, 8, 16 and 32
File system: Parallel file system	Coordinator processes: 1, 2, 4, 8 and 16
Network: Ethernet 1 Gbps / 10 Gbps	Processing per domain: 500 MIs

Table 7.3: Environment configuration of HPC model experiments using a many-core node



(a) Network of 1 Gbps

(b) Network of 10 Gbps

Figure 7.12: Simulation of HPC model using a many-core architecture

This architecture does not provide a significant improvement of performance when more CPU cores are used. Figure 7.12 shows that, besides there is a light increasing of performance, it is not proportional with the number of CPU cores used. Only when the number of CPU cores is increased from 1 to 2, and from 256 to 512, a light increasing of performance can be appreciated.

Using a faster network has different effects on those tests, depending on the number of I/O servers used. When the experiment uses only 1 I/O server, the overall system performance is practically the same using both networks. Otherwise, when several I/O servers are used, the 10 Gbps network provides a greater performance (see figure 7.13).

Finally, increasing the number of I/O servers provides a notable increasing of performance. It can be appreciated using both networks. However, the difference of performance is greater in those configurations that use a 10 Gbps network. Therefore, the overall system performance in this scenario is improved when the number of I/O nodes increases, because the level of parallelism in the I/O system increases as well.

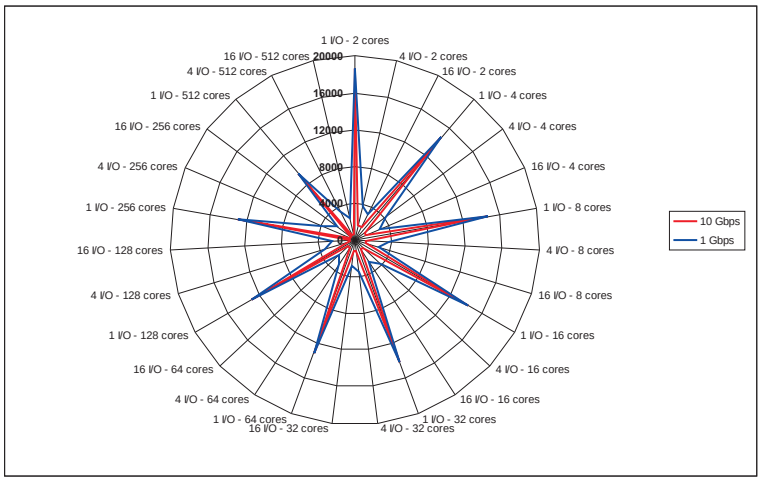


Figure 7.13: Simulation time of HPC experiments using a many-core node and different networks

7.4.2 Simulating the checkpointing application model

This section describes simulation experiments of the checkpointing model in two different environments. Therefore, the purpose of those experiments is to measure the throughput obtained (in MB/s) using different configurations. Thence, the application model is simulated in order to perform 8192 checkpoints, which consist of a data set of 128 GB.

First environment is a HPC scenario that consists of 128 computing nodes (see figure 7.4). The configuration for those experiments is described in table 7.4.

Environment model configuration	Application model configuration
Racks: 2 Boards per rack: 8 Computing nodes per board: 8 Total number of computing nodes: 128 CPU cores per node: 1, 4 and 16 CPU core speed: 20000 MIPS Storage nodes: 1, 4 and 8 File system: Parallel file system Network: Ethernet 1 Gbps / 10 Gbps	Number of checkpoints: 8192 Size of the state written to disk: 16 MB Processing per iteration: 100000 MIs

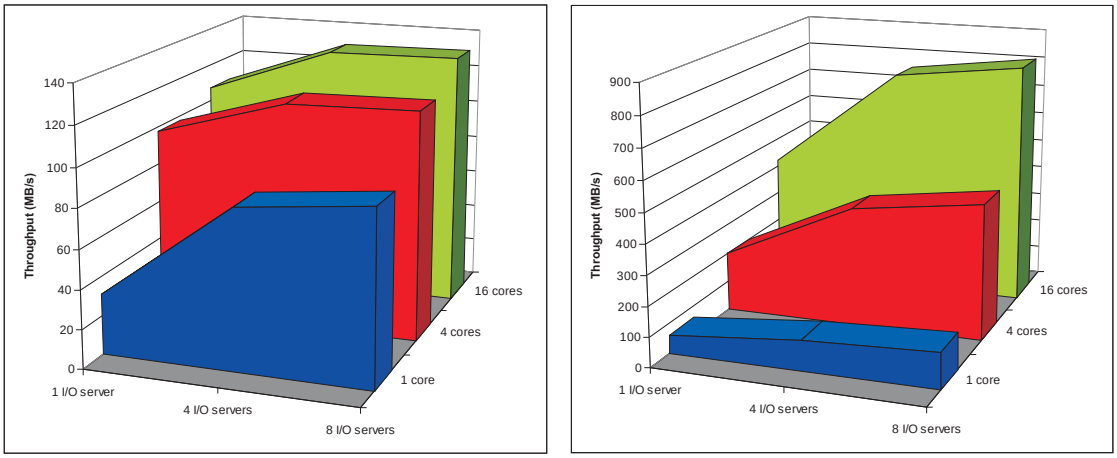
Table 7.4: Configuration of checkpointing model experiments using 128 nodes

Figure 7.14 shows that when a fast network is used (10 Gbps) the overall system performance scales proportionally with the number of CPU cores used. Otherwise, using a network of 1 Gbps this performance remains fixed when the number of CPU cores reaches 4. This is caused because the level of parallelism in the processing phases increases, obtaining a better performance. The reason why this performance remains fixed when more than 4 CPU cores per node are used is that the network acts as a system bottleneck. Then, the performance gained parallelizing the processing phase is lost when processes have to write their states to disk. It can be appreciated in figure 7.14(b). In some cases, using double CPU cores obtains almost double performance, which means that the system fully exploits

the bandwidth provided by the network.

Increasing the number of I/O servers also causes an improvement in the overall system performance. This improvement has a different impact, which depends directly of the configuration used. In those experiments that use a slow network, the only significant improvement is obtained when the number of I/O servers is increased from 1 to 4 using single-core CPUs. In the rest of the configurations, the difference of performance is almost insignificant (see figure 7.14(a)). It is caused due to the system are not able to make the most of those resources because the network acts as a system bottleneck. Otherwise, when a fast network is used, increasing the number of I/O servers produces an increasing of performance as well, which is greater when those servers increases from 1 to 4 independently of the number of CPU cores per node used (see figure 7.14(a)). This is caused because the processes fully exploit the faster network, using the bandwidth provided by the set of I/O servers in parallel.

Finally, figure 7.15 shows clearly that in those experiments the system bottleneck is the network. Using different networks in the same experiments have a direct impact on the overall system performance. This is caused because all processes write their states in parallel, using the network to write those states among the I/O servers, which causes a collapse in the network. When single-core CPUs are used, the performance obtained is practically the same for both networks. Otherwise, this difference of performance increases when the number of CPU cores per node increases as well, which means that increasing the number of processes executed in parallel make the most of the bandwidth provided by the faster network.



(a) Network of 1 Gbps

(b) Network of 10 Gbps

Figure 7.14: Throughput of checkpointing model using 128 nodes

Next experiments consist on simulating the checkpoint application model in a many-core node (see figure 7.11) using the configuration described in table 7.5.

Chart 7.16 shows that increasing the number of CPU cores produces a very slowly improvement of performance. Thence, using the double of CPU cores produces a slightly increasing of performance. The cause of this behavior is two-fold. First, the network acts as a system bottleneck when processes write their state to disk. Second, the network interface

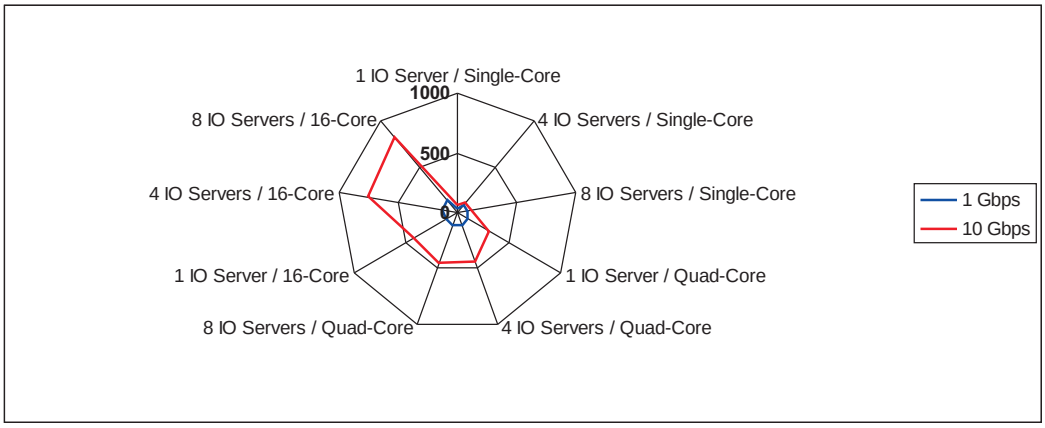


Figure 7.15: Throughput of checkpointing experiments using 128 nodes and different networks

Environment model configuration	Application model configuration
Computing nodes: 1 CPU cores per node: 2, 4, 8, 16, 32, 64, 128, 265 and 512 CPU core speed: 20000 MIPS Storage nodes: 1, 4 and 16 File system: Parallel file system Network: Ethernet 1 Gbps / 10 Gbps	Number of checkpoints: 8192 Size of the state written to disk: 16 MB Processing per iteration: 100000 MIs

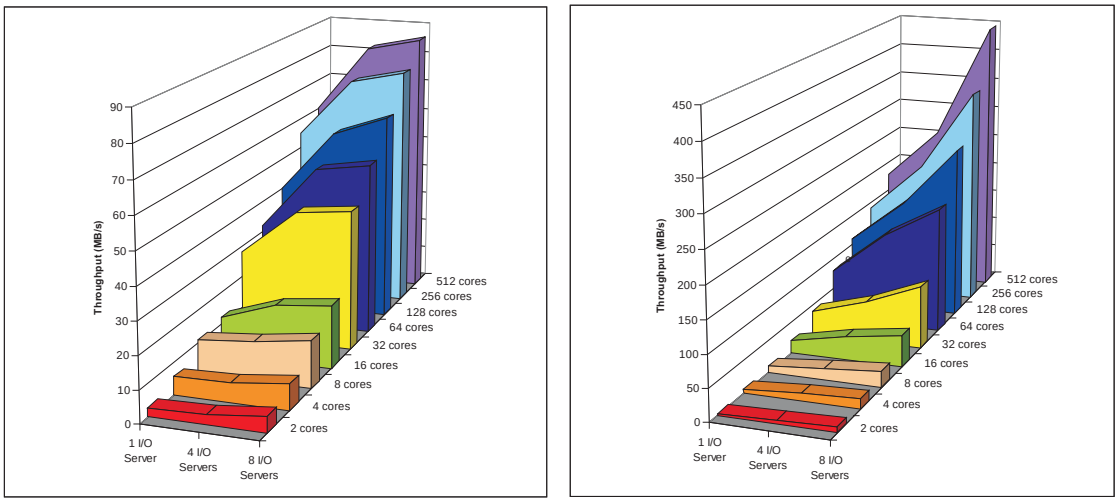
Table 7.5: Configuration of checkpointing model experiments using a many-core node

also acts as a system bottleneck, due to all processes uses the same interface for sending requests through the network. Therefore, although the level of parallelism increases when the number of CPU cores increases as well, the overall system performance suffers a drop of performance.

The throughput obtained using both networks is practically identical up to 16 CPU cores per node. When more than 32 CPU cores per node are used, the difference between those two networks can be clearly appreciated. Using a faster network lightly alleviates the drop of performance because the bandwidth of this network is fully exploited when the processes write their state to disk (see figure 7.17).

Similarly, increasing the number of I/O servers only has a direct impact on the overall system performance when the number of CPU cores per node is more than 16. It is caused because the level of parallelism in the I/O subsystem is fully exploited when the number processes that write in parallel increases as well, exploiting the bandwidth provided by a set of I/O servers in parallel. However, the greater difference of performance is noted when 4 I/O servers are used, because using 16 I/O servers does not provide a significant increasing of performance, which is caused because the system reach the maximum bandwidth provided by the network.

Both models of the HPC application and the checkpointing application show that many-core architectures provide lower performance than the cluster architecture composed by 128 computing nodes (see figure 7.4). This difference of performance is produced mainly because in the many-core, network interface acts as a system bottleneck.



(a) Network of 1 Gbps

(b) Network of 10 Gbps

Figure 7.16: Throughput of checkpoint model using a many-core architecture

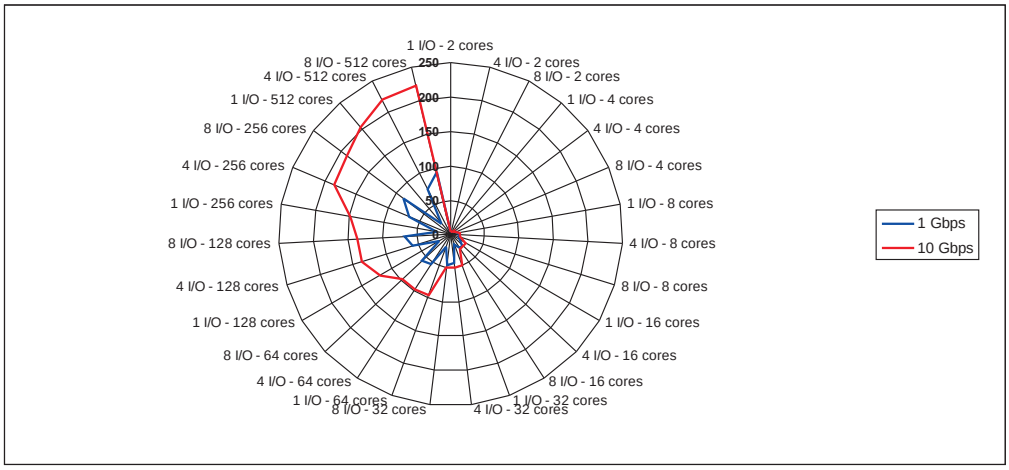


Figure 7.17: Throughput of checkpoint experiments using a many-core node and different networks

Those applications perform a massive number of I/O operations, which cause a continuous stress in the storage system. Thence, time required for performing computing and IPC is practically insignificant compared to the time required for reading and writing the corresponding data in the storage servers. Thus, those I/O requests sent from each process to the storage servers pass through the same network interface, and for the same switch. Otherwise, in the cluster architecture, each node shares the network interface, in the worst case, with 16 processes (when 16 CPU cores per node are used). Moreover, the cluster architecture uses a network of switches and several communication links, instead one switch and one communication link used in the many-core. Finally, the performance obtained by performing all the compute inside the same node (many-core node) is lost when the I/O operations are performed.

7.5 Measuring the performance of SIMCAN

In this section the performance results of the simulation itself for executing the experiments achieved in sections 7.4.1 and 7.4.2 are presented. Those simulations have been executed in a cluster, which features are described in table 7.6. The main purpose of this section is to calculate both the amount of time and memory needed for executing a specific simulation, depending of the size of the environment to be modeled, and the hardware resources available for executing each simulation.

Results obtained from those simulations are divided in two different sections, depending of the application model to be simulated: the HPC application model (see section 7.5.1) and the checkpointing application model (see section 7.5.2).

7.5.1 Performance of simulating the HPC application model

Figures 7.18 and 7.19 show the memory consumption and the execution time of the simulation experiments using a HPC architecture of 128 computing nodes (see figure 7.4). Those simulations have been executed using parallel simulations in a machine that contains 4 CPU cores and 4 GB of memory.

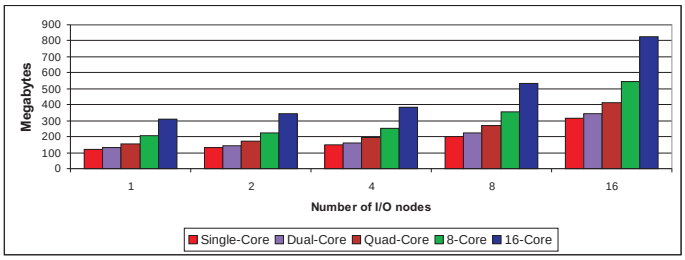


Figure 7.18: Memory consumption of HPC experiments using 128 nodes

Using the same network topology with different characteristics, like bandwidth and latencies, is totally independent both in the memory and time required for executing those simulations. This occurs because the calculations required for simulating different networks are the same.

Increasing the number of simulated CPU cores also increases the number of number of processes simulated simultaneously. Those processes send and receive messages through the simulated network. Then, the greater number of simulated processes, the greater number of

Number of nodes: 22
CPU Intel(R) Xeon(R) CPU E5405 @ 2.00GHz
4 GB of RAM memory
1 TB of Storage
Operating System Linux Debian. Kernel version 2.6.26-2-686
Network: Ethernet Gigabit

Table 7.6: Detailed features of the cluster where the simulations have been executed

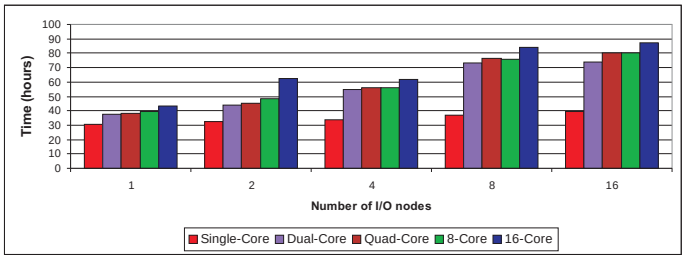


Figure 7.19: Execution time of HPC experiments using 128 nodes

messages sent through the simulated network, and the greater amount of memory and CPU processing needed. Similarly, increasing the number of I/O servers produces the same effect. In this case, each server manages a list of I/O requests. The more I/O servers simulated, the more lists of I/O requests to be managed, and the greater the amount of memory needed for storing those lists and CPU power for processing them.

Thus, increasing the number of CPU cores and I/O servers follows the same tendency in the amount of memory required. Otherwise, increasing the number of I/O servers requires more processing power than increasing the number of CPU cores per node.

Figures 7.20 and 7.21 show the memory consumption and the execution time of the simulation experiments using a HPC architecture of 1024 computing nodes (see figure 7.7). Those simulations have been executed using parallel simulations in 4 machines that contains 4 CPU cores (a total of 16 CPU cores) and 4 GB of memory each.

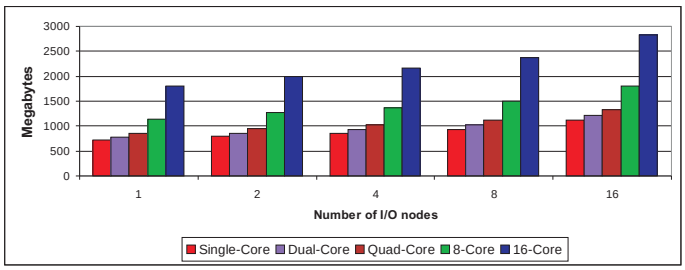


Figure 7.20: Memory consumption of HPC experiments using 1024 nodes

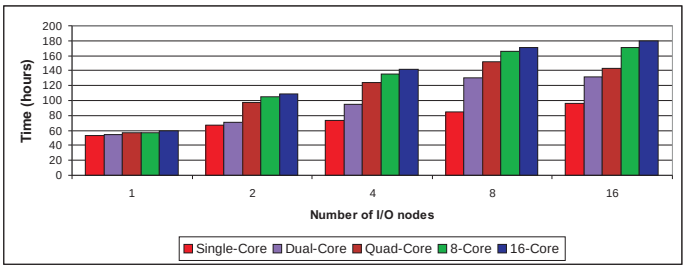


Figure 7.21: Execution time of HPC experiments using 1024 nodes

Figure 7.20 shows clearly that the memory needed for executing those experiments is directly related both to the number of I/O servers and the number of CPU cores per node.

From 1 to 4 CPU cores per node, the amount of memory needed is practically the same. When the number of CPUs per core reaches 8, then the amount of memory needed grows much faster. This is caused because the number of simulated processes also increases, and then the traffic generated by those processes in the simulated network, which require larger amounts of memory. The same goes with the number of I/O nodes. Thus, increasing the number of CPU cores and I/O servers follow almost the same tendency in the amount of memory required, growing faster when the number of CPUs per node increases.

Otherwise, increasing the number of I/O servers requires more processing power than increasing the number of CPU cores per node. This is caused because processing the lists of a set of I/O servers consumes more CPU power than simulating the requests sent through the simulated network. In this case, the time needed for executing the simulations grows faster when the number of I/O servers increases, than when the number of CPUs per node increases.

Finally, figures 7.22 and 7.23 show the memory consumption and the execution time of the simulation experiments using a HPC architecture of 1024 computing nodes (see figure 7.11). Those experiments have been executed in a machine using 4 CPU cores and 4 GB of memory (see table 7.6).

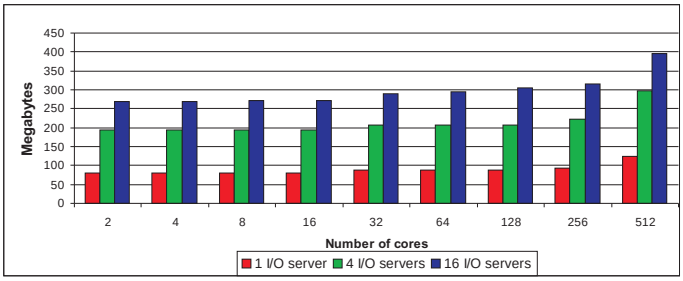


Figure 7.22: Memory consumption of HPC experiments using a many-core architecture

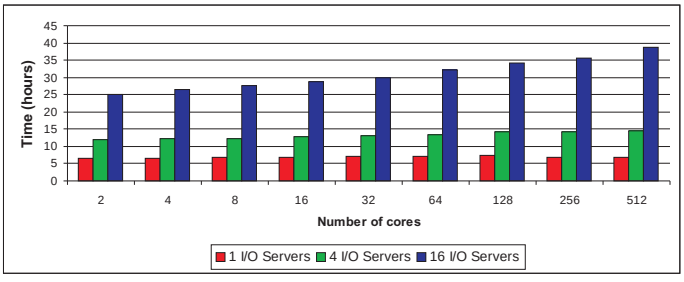


Figure 7.23: Execution time of HPC experiments using a many-core architecture

Those experiments require practically the same amount of memory for simulating the range of CPU cores per node from 1 to 256. Only using 512 CPU cores per node requires a larger amount of memory. This is caused by the extra traffic simulated in the network. Otherwise, changing the number of I/O servers produces a substantial increasing of memory. The same goes with the time needed for executing the simulations, which follows the same tendency that the memory consumption.

In order to compare the simulation time and memory consumption of different environments, figures 7.24 and 7.25 summarizes the performance results for those simulations that use 16 I/O servers.

Figure 7.24 shows the memory consumption depending of the number of simulated CPU cores. The memory required for executing those simulations remains practically identical from 2 CPU cores to 128 CPU cores. From 128 to 512 CPU cores, both many-core and 128 nodes architecture require almost the same amount of memory. Once the simulations reach 1024 computing nodes, the amounts of memory required are much larger. For example, in order to simulate 1024 CPU cores using the 128 nodes architecture, 544 MB of memory is needed, while simulating 1024 CPU cores using the 1024 nodes architecture requires 1120 MB of memory.

Figure 7.25 shows the execution time for simulating each architecture. Similarly to consumption of memory, when the number of nodes to be simulated increases, execution time increase as well. The same goes with the number of CPU cores. Using the many-core architecture, simulating the same application using a dual-core CPU requires 25 hours of execution, while the same application requires 38 hours when 512 CPU cores are used. For the 128 nodes architecture, the execution time required for executing the simulations when the number of simulated CPU increases, grows more slowly than for the 1024 nodes architecture. This is caused by the time spent on processing the high number of messages interchanged between the simulated processes.

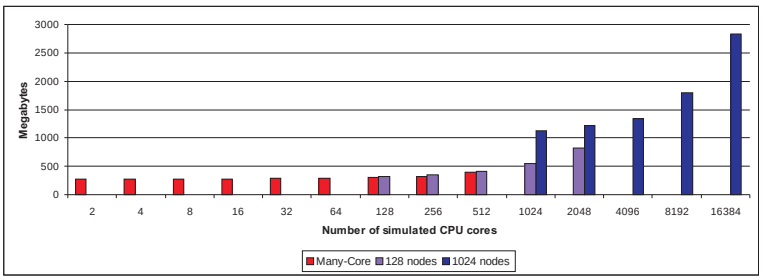


Figure 7.24: Memory consumption of HPC experiments using 16 I/O servers

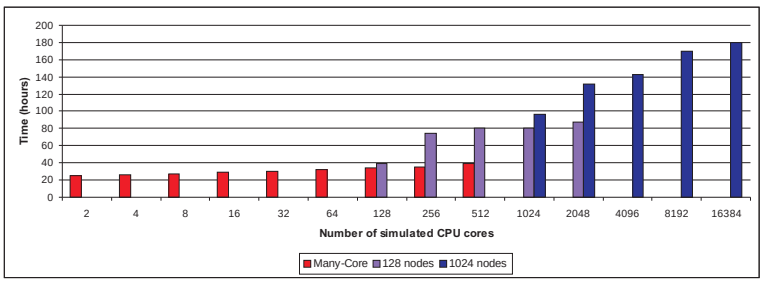


Figure 7.25: Execution time of HPC experiments using 16 I/O servers

7.5.2 Performance of simulating the checkpointing application model

Figures 7.26 and 7.27 show the memory consumption and the execution time of the simulation experiments using a HPC architecture of 128 computing nodes (see figure 7.4). Those experiments have been executed in a machine using 4 CPU cores and 4 GB of memory.

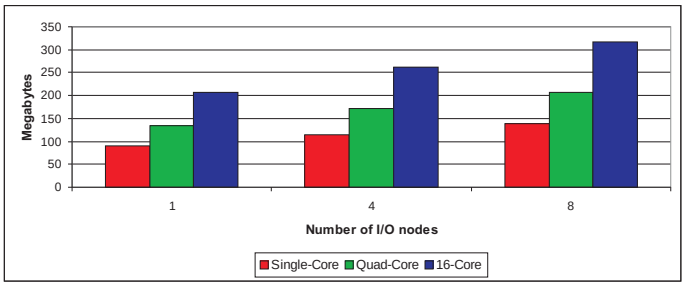


Figure 7.26: Memory consumption of checkpointing experiments using 128 nodes

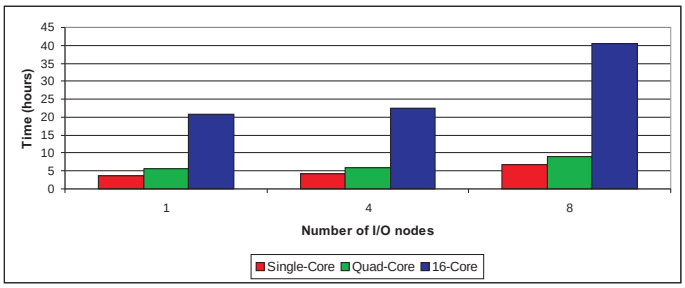


Figure 7.27: Execution time of checkpointing experiments using 128 nodes

First chart shows clearly that the amount of memory required for executing those simulations increase when the number of CPU cores per node increases as well. It is caused because the number of requests existent in the network also increases due to the number of simulated processes executed in parallel. Otherwise, increasing the number of I/O servers does not have a significant increasing of memory.

The amount of time needed for simulating those experiments is practically identical when 1 and 4 CPU cores per node are simulated. When 16 CPU cores per node are simulated, the amount of time needed for executing those simulations grows considerably. It is caused because the number of I/O operations produces large lists of requests in I/O servers, which require a high CPU processing time. The same goes when the number of simulated I/O server increases. Using 1 and 4 I/O servers, the time needed for simulating those experiments is similar, but using 8 I/O servers requires almost double amount of time for simulating those experiments.

Finally, figures 7.28 and 7.29 show the memory consumption and the execution time of the simulation experiments using a many-core node (see figure 7.11). Those experiments have been executed in a machine using 4 CPU cores and 4 GB of memory.

First chart shows that the amount of memory required for executing those simulations is practically identical for those simulations up to 128 CPU cores per node. The amount of

memory for executing those simulations lightly grows for those configurations that use more than 128 CPU core nodes per node. The same goes when the number of I/O nodes increases, but in this case, the amount of memory needed is directly related to the number of I/O servers simulated. Moreover, the amount of time needed for simulating those experiments follows almost the same tendency that the memory consumption. However, the time needed for executing those simulations grows when the number of CPU cores per node and I/O servers increases as well.

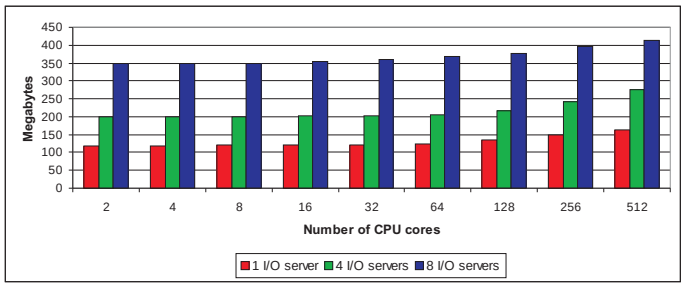


Figure 7.28: Memory consumption of checkpointing experiments using a many-core architecture

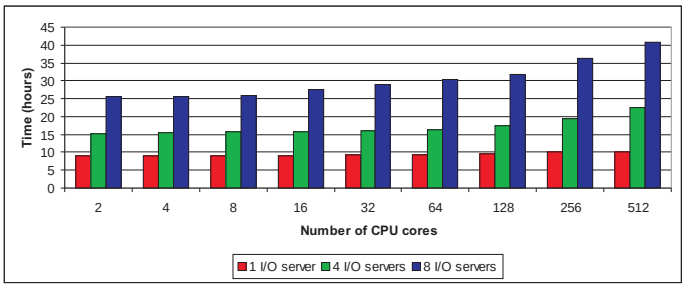


Figure 7.29: Execution time of checkpointing experiments using a many-core architecture

7.6 Summary

In this chapter, a set of experiments in order to demonstrate both the scalability and performance of the SIMCAN simulation platform has been fulfilled. Those experiments have been grouped in two different categories, depending of the purpose pursued.

In one hand, scalability experiments defined in this chapter are focused on simulating environments by increasing both the size of the problem and the size of the environment. The main purpose of those experiments is evaluating and analyzing how evolves both scalability and bottlenecks produced on a typical HPC multi-core architecture using different configurations.

In the other hand, performance experiments are focused on calculating both the amount of time and memory needed for executing a concrete simulation, depending of the size of the environment to be modeled, and the resources available for executing such simulation.

In conclusion, both models of HPC application and checkpointing application present a great increasing of performance in the three previous architectures (see figures 7.4, 7.7, and 7.11) when a fast network is used. This improvement of performance is produced mainly because the network acts as a system bottleneck, specifically in the storage system.

Increasing both the number of I/O servers and CPU cores per node provide a increasing of performance, which depends both of the architecture and configuration used. Generally, this performance remains fixed when the application raises the maximum throughput provided by the system configuration. Then, even increasing the resources like CPU cores and I/O servers does not warranty an increasing of performance.

Finally, the worst architecture for both application models is clearly the many-core node. The reason of its poor performance is that all processes executed in the many-core share the same network interface, which causes a significant bottleneck in the system.

Chapter 8

Conclusions and future works

Following the results achieved in this work, it's possible to say that the objectives presented at the beginning of this thesis have been successfully accomplished. In order to provide a detailed description of this statement, the main contributions of this thesis will be described in detail. Next, the publications achieved during the development of this thesis will be shown. Finally, several future works that are currently open to new research are described.

8.1 Main contributions

The main results of this thesis can be categorized in three main groups. First, a simulation platform using strategies for modeling parallel and distributed architectures. Second, strategies provided for modeling and simulating the execution of high performance computing applications. Finally, architecture optimizations that provide the best performance for a specific set of applications.

8.1.1 A simulation platform using strategies for modeling parallel and distributed architectures

The subsequent contributions provided in this section are enumerated below.

- A **new simulation platform** specifically designed for modeling parallel and distributed architectures has been proposed. We considered this approach to be a better choice that using generic simulation platforms or custom-made simulators. Moreover, this simulation platform has been used as a recipient for all the contributions proposed in this thesis.
- The simulation platform and the strategies proposed in this work allow to **simulate large-scale systems** by using a modular approach that allows to build a simulation environment by connecting and configuring together spare elements like nodes, bridges, disks, CPU, file systems, etc.
- **Good compromise** between **accuracy** and **performance**, and **flexibility** provided by the simulation platform for building a wide range of architectures with different configurations.

- A technique for **automatically splitting a model** in several domains for performing parallel simulations. Thus, users can easily perform parallel simulations of large models increasing the performance of such simulations.
- Strategies for **modeling the storage system**, which provide a general schema for building a wide range of I/O architectures. The major contribution of this point is a proposal for simulating file systems, which consists on using statistical models of the data block distribution of the file system to be simulated. An approach for simulating a generic model of parallel file systems is also described. Our platform also allows several methods to model the access data time of the disk drive units, including a proposal for calculating the access time using a linear interpolation-based technique.
- The ability of this simulation platform for simulating and **modeling new HPC environments**, and to evaluate the evolution of both scalability and bottlenecks existent in those environments by using different configurations and application models.
- Performance results of the simulation itself for the execution of the corresponding experiments have been obtained. The main purpose of this process is to **calculate both the amount of time and memory** needed for executing a specific simulation, depending of the size of the environment to be modeled, and the hardware resources available for executing each simulation.
- Finally, a graphic tool that provides a **fast an easy method for creating simulated environments**, specifically for novel users.

8.1.2 Strategies provided for modeling and simulating the execution of high performance computing applications

The subsequent contributions provided in this section are enumerated below.

- A proposal for **modeling the behavior of parallel applications** has been made. This proposal uses generic models and application patterns, which can be parameterized by the user to model a specific application that fits into such patterns. Proposed strategies are focused on covering from basic draft models of the applications to complete ports of the actual code.
- The **behavior of typical HPC applications and the behavior of typical checkpointing applications have been modeled**. Those models are targeted to analyze the performance of the storage system provided by the underlying architecture, due to those applications performing I/O operations massively.

8.1.3 Architecture optimizations to provide the best performance for a specific set of applications

The subsequent contribution provided in this section is described below.

- Experiments for **evaluating and analyzing** the evolution of both scalability and **bottlenecks existent on a typical HPC multi-core architecture** using different configurations have been fulfilled. Basically those experiments consist on executing the two previously described application models (HPC and checkpointing

applications) in several HPC architectures. Due to the nature of involved applications, those studies are focused mainly in the storage system. This relevance of the storage system lies in the time needed for executing the requested I/O operations by the involved processed, which makes practically insignificant the time spent on performing computing and IPC operations.

8.2 Publications related with this thesis

During the development of this thesis, some papers describing the results obtained in this work have been published. Those publications belong to journals and conferences, both national and international, and book chapters. Those publications are the following:

- Journals:
  - Alberto Núñez, Javier Fernández, Jose D. García, Félix García, and Jesús Carretero. *New techniques for simulating high performance MPI applications on large storage networks* [NnFG+10], pp. 40-57, Journal of Supercomputing, 51(1), 2010.
- Conferences:
  - Alberto Núñez, Javier Fernández and Jesús Carretero. *New Contributions for Simulating Large Distributed Systems.* [NFC10] pp. 227–230, DS-RT 2010, The 14th IEEE/ACM International Symposium on Distributed Simulation and Real Time Applications. Fairfax, Virginia, USA, October 2010.
  - Javier Fernández, Liangxiu Han, Alberto Núñez, Jesús Carretero, and Jano van Hemert. *Using architectural simulation models to aid the design of data intensive application* [FHN+09], pp. 163-168, ADVCOMP'09, Third International Conference on Advanced Engineering Computing and Applications in Sciences. Sliema, Malta, October 2009.
  - Alberto Núñez, Javier Fernández, Jose D. García and Jesús Carretero. *New techniques for simulating high performance MPI applications on large storage networks* [NFGC08b], pp. 444-452, IEEE Cluster 2008. Tsukuba, Japan. October, 2008.
  - Alberto Núñez, Javier Fernández, Jose D. García and Jesús Carretero. *Analyzing Scalable High-Performance I/O Architectures* [NFGC08a], pp. 631-637, PDPTA'08, The 2008 International Conference on Parallel and Distributed Processing Techniques and Applications. Las Vegas, Nevada (USA) July, 2008.
  - Alberto Núñez, Javier Fernández, Jesús Carretero, J. D. García and Laura Prada. *New Techniques for Modelling File Data Distribution on Storage Nodes* [NFG+08], pp. 175-182, ANSS'08, 41th Annual Simulation Symposium. Otawa, Canada. April, 2008.
  - Alberto Núñez, Javier Fernández, Jesús Carretero, Jose D. García and Laura Prada. *SIMCAN: A Simulator Framework for Computer Architectures and Storage Networks* [NFC+08b]. SIMUTools 2008, 1st International Workshop on

OMNeT++ held within First International Conference on Simulation Tools and Techniques for Communications, Networks and Systems. Marseille, France. March, 2008.

- Alberto Núñez, Javier Fernández, Jesús Carretero y Jose Daniel García. *Nuevas Técnicas para Modelar la Distribución de los Datos de Ficheros en Nodos de Almacenamiento* [NFCG07], pp.455-462. II Congreso Español de Informática (CEDI 2007). XVIII Jornadas de Paralelismo. Zaragoza, Spain, September, 2007. Vol I.

- Book chapters:

  - Alberto Núñez, Javier Fernández, and Jesús Carretero. Science and Supercomputing in Europe. Book chapter, *SIMCAN: A Highly Configurable Simulation Framework for HPC Architectures and Applications* [NFC08a], pp. 248-256, Monograf S.R.L., 2008.

8.3 Future works

This thesis has not only achieved the objectives proposed, but also let open some questions for future works that can be fulfilled. Some of them are the following:

- Proposing strategies for modeling the interaction between the CPU and the memory system in multi-core computers, without requiring executing applications at instruction level detail.
- Proposing strategies for estimating the time needed for executing a simulation model given the size of such model and the amount of resources available.
- Proposing new models for modeling and simulating cloud computing systems.
- Proposing new strategies for achieving the I/O performance in current systems like multi-core systems, using the simulation platform proposed in this work to check the validation of such strategies.
- Implementing new network models like hyperTransport and Ethernet 100 Gbps.

Bibliography

[AB02] R. Amicel and F. Bodin. Mastering startup costs in assembler-based compiled instruction-set simulation. In *Sixth Annual Workshop on Iinteraction between Compilers and Computer Architectures*, pages 39–44, 2002.

[ABB+00] V.S. Adve, R. Bagrodia, J.C. Browne, E. Deelman, A. Dube, E.N. Houstis, J.R. Rice, R. Sakellariou, D.J. Sundaram-Stukel, P.J. Teller, and M.K. Vernon. POEMS: end-to-end performance design of large parallel adaptive computational systems. *IEEE Transactions on Software Engineering*, 26(11):1027–1048, November 2000.

[ABF+10] L. Adhianto, S. Banerjee, M. Fagan, M. Krentel, G. Marin, J. Mellor-Crummey, and N. R. Tallent. HPCToolkit: tools for performance analysis of optimized parallel programs. *Concurrency and Computation: Practice and Experience*, 22(6):685–701, 2010.

[AC06] L. Adhianto and B. Chapman. Performance modeling of communication and computation in hybrid MPI and openMP applications. In *ICPADS'06: Parallel and Distributed Systems*, volume 2, pages 6 pp.–, 2006.

[ACG+07] D. August, J. Chang, S. Girbal, D. Gracia-Perez, G. Mouchard, D. Penry, O. Temam, and N. Vachharajani. UNISIM: An open simulation environment and library for complex architecture design and collaborative development. *Computer Architecture Letters*, 6(2):45–48, July-December 2007.

[AFF+09] Eduardo Argollo, Ayose Falcón, Paolo Faraboschi, Matteo Monchiero, and Daniel Ortega. COTSon: infrastructure for full system simulation. *SIGOPS Operating Systems Review*, 43(1):52–61, 2009.

[AG89] G. S. Almasi and A. Gottlieb. *Highly parallel computing*. Benjamin-Cummings Publishing Co., Inc., Redwood City, CA, USA, 1989.

[ALE02] Todd Austin, Eric Larson, and Dan Ernst. SimpleScalar: An Infrastructure for Computer System Modeling. *Computer*, 35(2):59–67, 2002.

[ALP03] J. González A. Loureiro and T.F. Pena. A parallel 3D semiconductor device simulator for gradual heterojunction bipolar transistors. *Journal of Numerical Modelling: electronic networks, devices and fields*, 16:53–66, 2003.

[Amd67] Gene M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of AFIPS Conference*, 1967.

[arm08] ARM Ltd. Realview developer suit, 2008. http://www.arm.com/products/ /DevTools/RealViewDevSuite.html.

[AS00] Vikram Adve and Rizos Sakellariou. Application representations for multi-paradigm performance modeling of large-scale parallel scientific codes. *International Journal of High Performance Computing Applications*, 14(4):304–316, November 2000.

[AV04] Vikram S. Adve and Mary K. Vernon. Parallel program performance prediction using deterministic task graph analysis. *ACM Transactions on Computer Systems (TOCS)*, 22(1):94–136, February 2004.

[BBE+99] Sandeep Bajaj, Lee Breslau, Deborah Estrin, Kevin Fall, Sally Floyd, Padma Haldar, Mark Handley, Ahmed Helmy, John Heidemann, Polly Huang, Satish Kumar, Steven McCanne, Reza Rejaie, Puneet Sharma, Kannan Varadhan, Ya Xu, Haobo Yu, and Daniel Zappala. Improving simulation for network research. Technical Report 99-702b, University of Southern California, March 1999.

[BC05] Daniel P. Bovet and Marco Cesati. *Understanding the Linux Kernel*. O'Reilly Media, November 2005.

[BDDP99] Rajive Bagrodia, Ewa Deelman, Steven Docy, and Thomas Phan. Performance prediction of large parallel applications using parallel simulations. *ACM SIGPLAN Notices*, 34(8):151–162, August 1999.

[BDH+06] N. L. Binkert, R. G. Dreslinski, L. R. Hsu, K. T. Lim, A. G. Saidi, and S. K. Reinhardt. The M5 simulator: Modeling networked systems. *IEEE Micro*, 26(4):52–60, July-August 2006.

[BDK97] R. Bagrodia, S. Docy, and A. Kahn. Parallel simulation of parallel file systems and I/O programs. In *Proceedings of the 1997 ACM/IEEE conference on Supercomputing*, San Jose, CA, USA, 1997.

[BDP01] Rajive Bagrodia, Ewa Deelman, and Thomas Phan. Parallel simulation of large-scale parallel applications. *International Journal of High Performance Computing Applications*, 15(1):3–12, February 2001.

[BDV94] G. Burns, R. Daoud, and J. Vaigl. LAM: An open cluster environment for MPI. In *Proceedings of Supercomputing Symposium '94*, pages 379–386, 1994.

[Bed90] Robert Bedichek. Some efficient architecture simulation techniques. In *Winter 1990 USENIX Conference*, pages 53–63, Berkeley, California, USA, 1990.

[Bed95] Robert C. Bedichek. Talisman: Fast and accurate multicomputer simulation. In *Proceedings of the 1995 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 14–24, 1995.

[Bha97] P. Bhargava. MPI-LITE user manual, Release 1.1. Technical report, Parallel Computing Lab, University of California, 1997.

[BMG07] Darius Buntinas, Guillaume Mercier, and William Gropp. Implementation and evaluation of shared-memory communication and synchronization operations in MPICH2 using the nemesis communication subsystem. *Parallel Computing*, 33(9):634–644, 2007.

[BMP+04] Greg Bronevetsky, Daniel Marques, Keshav Pingali, Peter Szwed, and Martin Schulz. Application-level checkpointing for shared memory programs. *ACM SIGARCH Computer Architecture News*, 32(5):235–247, October 2004.

[BMT+98] Rajive Bagrodia, Richard Meyer, Mineo Takai, Yu an Chen, Xiang Zeng, Jay Martin, and Ha Yoon Song. Parsec: A parallel simulation environment for complex systems. *Computer Magazine*, 31(10):77–85, October 1998.

[Boc05] The Bochs project. Bochs: The cross platform IA-32 emulator, August 2005. http://bochs.sourceforge.net.

[BSD95] H. J. C Berendsen, D. Van Der Spoel, and R. Van Drunen. GROMACS: A message-passing parallel molecular dynamics implementation. *Computer Physics Communications*, 91:43–56, September 1995.

[BSSG08] John S. Bucy, Jiri Schindler, Steven W. Schlosser, and Gregory R. Ganger. *The DiskSim Simulation Environment Version 4.0 Reference Manual*. Dept. Electrical and Computer Engineering, Carnegie Mellon University, Pittsburgh, PA 15213, May 2008.

[BYP+91] M. Butler, Tse-Yu Yeh, Y. Patt, M. Alsup, H. Scales, and M. Shebanow. Single instruction stream parallelism is greater than two. In *The 18th Annual International Symposium on Computer Architecture*, pages 276–286, 1991.

[CCC+03] Félix García Carballeira, Alejandro Calderón, Jesús Carretero, Javier Fernández, and Jose M. Pérez. The design of the Expand parallel file system. *International Journal of High Performance Computing Applications*, 17(1):21–37, 2003.

[CDK+00] Rohit Chandra, Leo Dagum, Dave Kohr, Dror Maydan, Jeff McDonald, and Ramesh Menon. *Parallel Programming in openMP*. Morgan Kaufmann, 1st edition, 2000.

[CF96] Peter F. Corbett and Dror G. Feitelson. The Vesta parallel file system. *ACM Transactions on Computer Systems*, 14(3):225–264, 1996.

[Cha99] Xinjie Chang. Network simulations with OPNET. In *WSC'99: Proceedings of the 31st conference on Winter simulation*, pages 307–314, New York, NY, USA, 1999. ACM.

[CIRT00] P. H. Carns, W. B. Ligon III, R. B. Ross, and R. Thakur. PVFS: A parallel file system for Linux clusters. In *Proceedings of the 4th Annual Linux Showcase and Conference*, pages 317–327, Atlanta, GA, October 2000.

[CK94] Bob Cmelik and David Keppel. Shade: a fast instruction-set simulator for execution profiling. In *SIGMETRICS'94: Proceedings of the 1994 ACM SIGMETRICS conference on Measurement and modeling of computer systems*, pages 128–137, New York, NY, USA, 1994. ACM.

[CK05] Christophe Cérin and Michel Koskas. Mining traces of large scale systems. In *ICA3PP'05: International Conference on Algorithms and Architectures*, pages 132–138, 2005.

[CL99] Computing-Laboratory. *The JavaSim User's Manual*. Department of Computing Science, University of Newcastle upon Tyne, 1999.

[CMMS88] R. C. Covington, S. Madala, V. Mehta, and J. R. Jump J. B. Sinclair. The rice parallel processing testbed. *ACM SIGMETRICS Performance Evaluation Review*, 16(1):4–11, 1988.

[CRR+07a] Felipe Cabarcas, Alejandro Rico, David Ródenas, Xavier Martorell, Álex Ramírez, and Eduard Ayguade. CellSim: A validated modular heterogeneus multiprocessor simulator. In *CEDI'07: II Congreso Español de Informática. XVIII Jornadas de Paralelismo*, pages 181–188, Zaragoza, Spain, September 2007.

[CRR+07b] Felipe Cabarcas, Alejandro Rico, David Ródenas, Xavier Martorell, Álex Ramírez, and Eduard Ayguade. Implementation and validation of a Cell simulator using UNISIM. In *3rd HiPEAC Industrial Workshop on Compilers and Architectures*, Haifa, Israel, 2007.

[CTT94] R. Card, T. Y. Ts'o, and S. Tweedie. Design and implementation of the second extended file system. In *Proceedings of the 1994 Amsterdam Linux Conference*, Amsterdam. The Netherlands, 1994.

[DB99] John R. Douceur and William J. Bolosky. A large-scale study of file-system contents. *ACM SIGMETRICS Performance Evaluation Review*, 27(1):59–70, June 1999.

[DBP+04] Timothy E. Denehy, John Bent, Florentina I. Popovici, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Deconstructing storage arrays. In *ASPLOS-XI: Proceedings of the 11th international conference on Architectural support for programming languages and operating systems*, pages 59–71, New York, NY, USA, 2004. ACM.

[DG08] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, January 2008.

[Dij59] Edsger W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1:269–271, 1959.

[DJS89] S. Dwarkadas, J. R. Jump, and J. B. Sinclair. Efficient simulation of cache memories. In *WSC '89: Proceedings of the 21st conference on Winter simulation*, pages 1032–1041, New York, NY, USA, 1989. ACM.

[DJS94] Sandhya Dwarkadas, J. Robert Jump, and James B. Sinclair. Execution-driven simulation of multiprocessors: Address and timing analysis. *ACM Transactions on Modeling and Computer Simulation*, 4(4):314–338, 1994.

[DLP03] Jack J. Dongarra, Piotr Luszczek, and Antoine Petitet. The LINPACK benchmark: Past, present, and future. *Concurrency and Computation: Practice and Experience*, 15:2003, 2003.

[DM84] J. K. Doyle and K. I. Mandelberg. A portable PDP-11 simulator. *Software, Practice and Experience*, 14(11):1047–1059, 1984.

[DSSP06] Peter DeRosa, Kai Shen, Christopher Stewart, and Jonathan Pearson. Realism and simplicity: disk simulation for instructional os performance evaluation. In *SIGCSE'06: Proceedings of the 37th SIGCSE technical symposium on Computer science education*, pages 308–312, New York, NY, USA, 2006. ACM.

[EdBN00] L. Eeckhout, K. de Bosschere, and H. Neefs. Performance analysis through synthetic trace generation. In *ISPASS'00: IEEE International Symposium on Performance Analysis of Systems and Software*, pages 1–6, 2000.

[FC88] Richard M. Fujimoto and William B. Campbell. Efficient instruction level simulation of computers. *Transactions of the Society for Computer Simulation International*, 5(2):109–124, 1988.

[FHN+09] Javier Fernández, Liangxiu Han, Alberto Núñez, Jesús Carretero, and Jano van Hemert. Using architectural simulation models to aid the design of data intensive application. In *ADVCOMP'09: International Conference on Advanced Engineering Computing and Applications in Sciences*, pages 163–168, Los Alamitos, CA, USA, 2009. IEEE Computer Society.

[FP94] J.W.C. Fu and J.H. Patel. Trace driven simulation using sampled traces. In *International Conference on System Sciences. Vol. I: Architecture*, pages 211–220, January 1994.

[FSI+07] Rosa Filgueira, David E. Singh, Florin Isaila, Jesús Carretero, and Antonio Garcia Loureiro. Optimization and evaluation of parallel I/O in BIPS3D parallel irregular application. In *IPDPS07: 21th International Parallel and Distributed Processing Symposium*, pages 1–8, 2007.

[FSS00] Thomas Fahringer, Bernhard Scholz, and Xian-He Sun. Execution-driven performance analysis for distributed and parallel systems. In *Workshop on Software and Performance*, pages 204–215, 2000.

[G84] Sargent R. G. *Simulation model validation. Chapter 19 in Simulation and Model-Based Methodologies: An Integrative View.* Heidelberg, Germany: Springer-Verlag, 1984.

[GFB+04] Edgar Gabriel, Graham E. Fagg, George Bosilca, Thara Angskun, Jack J. Dongarra, Jeffrey M. Squyres, Vishal Sahay, Prabhanjan Kambadur, Brian Barrett, Andrew Lumsdaine, Ralph H. Castain, David J. Daniel, Richard L.

Graham, and Timothy S. Woodall. Open MPI: Goals, concept, and design of a next generation MPI implementation. In *Proceedings of the 11th European PVM/MPI Users' Group Meeting*, pages 97–104, Budapest, Hungary, September 2004.

[GFP09] Marius Gligor, Nicolas Fournel, and Frédéric Pétrot. Using binary translation in event driven simulation for fast and flexible MPSoC simulation. In *CODES+ISSS '09: Proceedings of the 7th IEEE/ACM international conference on Hardware/Software codesign and system synthesis*, pages 71–80, New York, NY, USA, 2009. ACM.

[GHLL+98] W. Gropp, S. Huss-Lederman, A. Lumsdaine, E. Lusk, B. Nitzberg, W. Saphir, and M. Snir. *MPI: The Complete Reference*. MTI-Press, 2nd edition edition, 1998.

[GL98] William Gropp and Ewing Lusk. *Users's guide for MPE:Extensions for MPI programs*. Argonne National Laboratory, 1998.

[GL99] William Gropp and Ewing L. Lusk. Reproducible measurements of MPI performance characteristics. In *Proceedings of the 6th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 11–18, London, UK, 1999. Springer-Verlag.

[GLB00] Sergi Girona, Jesús Labarta, and Rosa M. Badia. Validation of Dimemas communication model for MPI collective operations. In *Proceedings of the 7th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 39–46, London, UK, 2000. Springer-Verlag.

[GLDS96] William Gropp, Ewing Lusk, Nathan Doss, and Anthony Skjellum. A high-performance, portable implementation of the MPI Message-Passing Interface standard. *Parallel Computing*, 22:789–828, 1996.

[GMM99] Somnath Ghosh, Margaret Martonosi, and Sharad Malik. Cache miss equations: A compiler framework for analyzing and tuning memory behavior. *ACM Transactions on Programming Languages and Systems*, 21:703–746, 1999.

[GO10] M. Gerndt and M. Ott. Automatic performance analysis with periscope. *Concurrency and Computation: Practice & Experience*, 22(6):736–748, 2010.

[GPF09] General parallel file system documentation, September 2009. Available online: http://publib.boulder.ibm.com/infocenter/clresctr/vxrx/topic//com.ibm.cluster.gpfs.doc/gpfsbooks.html.

[Gus88] John L. Gustafson. Reevaluating Amdahl's law. *Communications of ACM*, 31(5):532–533, 1988.

[GWW+10] Markus Geimer, Felix Wolf, Brian J. N. Wylie, Erika Ábrahám, Daniel Becker, and Bernd Mohr. The Scalasca performance toolset architecture. *Concurrency and Computation: Practice and Experience*, 22(6):702–719, 2010.

[Hac92] Anna Hac. Modeling distributed file systems. *ACM SIGMETRICS Performance Evaluation Review*, 19(4):22–27, May 1992.

[HGG+99] A. Halambi, P. Grun, V. Ganesh, A. Khare, N. Dutt, and A. Nicolau. EXPRESSION: a language for architecture exploration through compiler/simulator retargetability. In *DATE'99: Design, Automation and Test in Europe Conference and Exhibition*, pages 485–490, Munich, Germany, 1999.

[HLK03] Chao Huang, Orion Lawlor, and L. V. Kale. Adaptive MPI. In *LCPC'03: Proceedings of the 16th International Workshop on Languages and Compilers for Parallel Computing*, pages 306–322, College Station, Texas, United States, October 2003.

[HLR08] Thomas R. Henderson, Mathieu Lacage, and George F. Riley. Network simulations with the ns-3 simulator. In *SIGCOMM'08: Conference of the Special Interest Group on Data Communications*, page 527. ACM, August 2008.

[HPRA02] C.J. Hughes, V.S. Pai, P. Ranganathan, and S.V. Adve. Rsim: simulating shared-memory multiprocessors with ILP processors. *Computer*, 35(2):40–49, February 2002.

[HRFR06] Thomas R. Henderson, Sumit Roy, Sally Floyd, and George F. Riley. ns-3 project goals. In *WNS2'06: Proceeding from the 2006 workshop on ns-2: the IP network simulator*, page 13, New York, NY, USA, 2006. ACM.

[HSW+04] Nikolaos Hardavellas, Stephen Somogyi, Thomas F. Wenisch, E. Wunderlich, Shelley Chen, Jangwoo Kim, Babak Falsafi, James C. Hoe, and Andreas G. Nowatzyk. Simflex: A fast, accurate, flexible full-system simulation framework for performance evaluation of server architecture. *ACM SIGMETRICS Performance Evaluation Review*, 31:31–35, 2004.

[HYZ05] X. He, Q. K. Yang, and M. Zhang. SPEK: A storage performance evaluation kernel module for block-level storage systems under faulty conditions. *IEEE Transactions on Dependable and Secure Computing*, 2(2):138–149, April 2005.

[KAH+01] D. J. Kerbyson, H. J. Alme, A. Hoisie, F. Petrini, H. J. Wasserman, and M. Gittings. Predictive performance and scalability modeling of a large-scale application. In *Supercomputing '01: Proceedings of the 2001 ACM/IEEE conference on Supercomputing (CDROM)*, page 37, New York, NY, USA, 2001. ACM.

[KDH+05] J. A. Kahle, M. N. Day, H. P. Hofstee, C. R. Johns, T. R. Maeurer, and D. Shippy. Introduction to the Cell multiprocessor. *IBM Journal of Research and Development*, 49(4/5):589–604, 2005.

[KE91] D.R. Kaeli and P.G. Emma. Branch history table prediction of moving target branches due to subroutine returns. In *The 18th Annual International Symposium on Computer Architecture*, pages 34–42, 1991.

[Kes88] S. Keshav. REAL: A network simulator. Technical report, University of California at Berkeley, Berkeley, CA, USA, 1988.

[KK96] L.V. Kale and S. Krishnan. *Parallel Programming Using C++*. The MIT Press, July 1996.

[KK98] George Karypis and Vipin Kumar. MeTiS: A software package for partitioning unstructured graphs, partitioning meshes, and computing fill-reducing orderings of sparse matrices. Technical report, 1998.

[KRR04] M. Khnemann, T. Rauber, and G. Runger. A source code analyzer for performance prediction. In *18th International Parallel and Distributed Processing Symposium (CDROM)*. IEEE, April 2004.

[KWH02] D.J. Kerbyson, H.J. Wasserman, and A. Hoisie. Exploring advanced architectures using performance prediction. In *IWIA'02: International Workshop on Innovative Architecture for Future Generation High-Performance Processors and Systems*, pages 27–37, 2002.

[Lar93] J.R. Larus. Efficient program tracing. *Computer*, 26(5):52–61, May 1993.

[Lau94] G. Lauterbach. Accelerating architectural simulation by parallel execution of trace samples. In *International Conference on System Sciences. Vol. I: Architecture*, pages 205–210, January 1994.

[LC01] Eric Larson and Saugata Chatterjee. MASE: A novel infrastructure for detailed microarchitectural modeling. In *ISPASS'01: Proceedings of the 2001 International Symposium on Performance Analysis of Systems and Software*, pages 9 pp.–, November 2001.

[LDG+08] Mingsong Lv, Qingxu Deng, Nan Guan, Yaming Xie, and Ge Yu. ARMISS: An instruction set simulator for the arm architecture. In *ICESS '08: International Conference on Embedded Software and Systems*, pages 548–555, July 2008.

[LDK+05] Piotr Luszczek, Jack J. Dongarra, David Koester, Rolf Rabenseifner, Bob Lucas, Jeremy Kepner, John Mccalpin, David Bailey, and Daisuke Takahashi. Introduction to the HPC Challenge Benchmark Suite. Technical report, 2005.

[LGP+96] Jesús Labarta, Sergi Girona, Vincent Pillet, Toni Cortes, and Luis Gregoris. DiP: A parallel program development environment. In *Euro-Par'96*, volume 2, pages 665–674, Lyon, France, August 1996.

[LM93] M. C. Little and D. L. McCue. Construction and use of a simulation package in C++. Technical Report 437, Department of Computing Science. University of Newcastle upon Tyne, July 1993.

[LM94] M.C. Little and D. McCue. Construction and use of a simulation package in C++. *C User's Journal*, 12(3), 1994. Postscript file available at http://cxxsim.ncl.ac.uk/homepage.html.

[LPI88] S. Laha, J.H. Patel, and R.K. Iyer. Accurate low-cost methods for performance evaluation of cache memory systems. *IEEE Transactions on Computers*, 37(11):1325–1336, November 1988.

[MAF91] C. Mills, S.C. Ahalt, and J. Fowler. Compiled instruction set simulation. *Software, Practice and Experience*, 21(8):877–889, 1991.

[Mag93] Peter S. Magnusson. A design for efficient simulation of a multiprocessor. In *MASCOTS '93: Proceedings of the International Workshop on Modeling, Analysis, and Simulation on Computer and Telecommunication Systems*, pages 69–78, San Diego, CA, USA, 1993. Society for Computer Simulation International.

[MANR09] J. Miguel-Alonso, J. Navaridas, and F. J. Ridruejo. Interconnection network simulation using traces of MPI applications. *International Journal of Parallel Programming*, 37(2):153–174, 2009.

[May87] C. May. Mimic: a fast system/370 simulator. In *SIGPLAN '87: Papers of the Symposium on Interpreters and interpretive techniques*, pages 1–13, New York, NY, USA, 1987. ACM.

[MBC+06] José Moreira, Michael Brutman, José Castaños, Thomas Engelsiepen, Mark Giampapa, Tom Gooding, Roger Haskin, Todd Inglett, Derek Lieber, Pat McCarthy, Mike Mundy, Jeff Parker, and Brian Wallenfelt. Designing a highly-scalable operating system: the Blue Gene/L story. In *Conference on High Performance Networking and Computing. Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, page Article No. 118, Tampa, Florida, USA, November 2006. ACM.

[McD91] Chris McDonald. A network specification language and execution environment for undergraduate teaching. In *ACM Computer Science Education Technical Symposium*, pages 25–34, March 1991.

[MCE+02] P.S. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hallberg, J. Hogberg, F. Larsson, A. Moestedt, and B. Werner. Simics: A full system simulation platform. *Computer*, 35(2):50–58, February 2002.

[MHW02] Carl J. Mauer, Mark D. Hill, and David A. Wood. Full-system timing-first simulation. In *SIGMETRICS '02: Proceedings of the 2002 ACM SIGMETRICS international conference on Measurement and Modeling of computer systems*, pages 108–116, New York, NY, USA, 2002. ACM.

[Mit03] Michael Mitzenmacher. Dynamic models for file sizes and double pareto distributions. *Internet Math*, 1(3):305–333, 2003.

[MN05] Alexander Muzy and James J. Nutaro. Algorithms for efficient implementations of the DEVS & DSDEVS abstract simulators. In *Proceedings of the 1st Open International Conference on Modeling & Simulation*, pages 401–407, ISIMA/Blaise Pascal University, 2005.

[MSB+05] Milo M. K. Martin, Daniel J. Sorin, Bradford M. Beckmann, Michael R. Marty, Min Xu, Alaa R. Alameldeen, Kevin E. Moore, Mark D. Hill, and David A. Wood. Multifacet's general execution-driven multiprocessor simulator (GEMS) toolset. *ACM SIGARCH Computer Architecture News*, 33(4):92–99, November 2005.

[MSDS10] Hans Meuer, Erich Strohmaier, Jack Dongarra, and Horst D. Simon. Top500 supercomputer sites, 2010. http://www.top500.org.

[MSM+01] José Carlos Mouriño, David E. Singh, María J. Martín, J. M. Eiroa, Francisco F. Rivera, Ramon Doallo, and Javier D. Bruguera. Parallelization of the STEM-II air quality model. In *HPCN Europe'01: 9th International Conference on High-Performance Computing and Networking*, pages 543–546, 2001.

[MW95] P. Magnusson and B. Werner. Efficient memory simulation in SimICS. In *SS'95: Proceedings of the 28th Annual Simulation Symposium*, pages 62–73, Washington, DC, USA, 1995. IEEE Computer Society.

[NFC08a] Alberto Núñez, Javier Fernández, and Jesús Carretero. *Science and Supercomputing in Europe. Book chapter, SIMCAN: A Highly Configurable Simulation Framework for HPC Architectures and Applications. Pages: 248–256.* Monograf S.R.L., 2008.

[NFC+08b] Alberto Núñez, Javier Fernández, Jesús Carretero, Jose D. García, and Laura Prada. SIMCAN: A SIMulator Framework for Computer Architectures and Storage Networks. In *SIMUTools'08, First International Conference on Simulation Tools and Techniques for Communications, Networks and Systems*, pages 8 pp.–, Marseille, France, March 2008. ACM.

[NFC10] Alberto Núñez, Javier Fernández, and Jesús Carretero. New contributions for simulating large distributed systems. In *DS-RT 2010: The 14th IEEE/ACM International Symposium on Distributed Simulation and Real Time Applications*, pages 227–230, Fairfax, Virginia, USA, October 2010.

[NFCG07] Alberto Núñez, Javier Fernández, Jesús Carretero, and Jose Daniel García. Nuevas técnicas para modelar la distribución de los datos de ficheros en nodos de almacenamiento. In *CEDI'07: II Congreso Español de Informática. XVIII Jornadas de Paralelismo*, pages 455–462, Zaragoza, Spain, September 2007.

[NFG+08] Alberto Núñez, Javier Fernández, Jose D. García, Laura Prada, and Jesús Carretero. New Techniques for Modeling File Data Distribution on Storage Nodes. In *ANSS'08, 41st Annual Simulation Symposium*, pages 175–182, Ottawa, Canada, April 2008. IEEE.

[NFGC08a] Alberto Núñez, Javier Fernández, Jose D. García, and Jesús Carretero. Analyzing scalable High-Performance I/O architectures. In *PDPTA'08, The 2008 International Conference on Parallel and Distributed Processing Techniques and Applications*, pages 631–637, Las Vegas, Nevada (USA), July 2008.

[NFGC08b] Alberto Núñez, Javier Fernández, Jose D. García, and Jesús Carretero. New techniques for simulating high performance MPI applications on large storage networks. In *IEEE Cluster 2008*, pages 444–452, Tsukuba, Japan, October 2008.

[NKP+00] G. R. Nudd, D. J. Kerbyson, E. Papaefstathiou, S. C. Perry, J. S. Harper, and D. V. Wilcox. Pace: A toolset for the performance prediction of parallel and distributed systems. *International Journal of High Performance Computing Applications*, 14(3):228–251, 2000.

[NnFG+10] Alberto Núñez, Javier Fernández, Jose D. Garcia, Félix Garcia, and Jesús Carretero. New techniques for simulating high performance MPI applications on large storage networks. *Journal of Supercomputing*, 51(1):40–57, 2010.

[Nol07] John Noll. COEN 177. Project 3: File system simulation. 2007.

[NS2] Network simulator NS-2. http://www.isi.edu/nsnam/ns.

[NTN06] Takashi Nakada, Tomoaki Tsumura, and Hiroshi Nakashima. Design and implementation of a workload specific simulator. In *ANSS '06: Proceedings of the 39th annual Simulation Symposium*, pages 230–243, Washington, DC, USA, 2006. IEEE Computer Society.

[Nut05] James J. Nutaro. Adevs (a discrete event system simulator) C++ library, 2005. http://www.ece.arizona.edu/nutaro/index.php.

[PB98] Sundeep Prakash and Rajive L. Bagrodia. MPI-SIM: using parallel simulation to evaluate MPI programs. In *Winter Simulation Conference Proceedings*, volume 1, pages 467–474, Washington, DC, USA, December 1998.

[PGK88] David A. Patterson, Garth Gibson, and Randy H. Katz. A case for redundant arrays of inexpensive disks (RAID). In *Proceedings of the 1988 ACM SIGMOD international conference on Management of data*, pages 109–116, Chicago, Illinois, United States, 1988.

[PK05] Bernd Page and W. Kreutzer. *The Java Simulation Handbook: Simulating Discrete Event Systems with UML and Java.* Shaker Verlag GmbH, 2005.

[PMA05] Francisco Javier Ridruejo Perez and José Miguel-Alonso. INSEE: An interconnection network simulation and evaluation environment. In *Euro-Par'05: 11th International Euro-Par Conference on Parallel Processing*, pages 1014–1023, Lisbon, Portugal, September 2005. Springer.

[PMP+04] Joshua J. Pieper, Alain Mellan, JoAnn M. Paul, Donald E. Thomas, and Faraydon Karim. High level cache simulation for heterogeneous multiprocessors. In *DAC '04: Proceedings of the 41st annual conference on Design automation*, pages 287–292, New York, NY, USA, 2004. ACM.

[PY93] D.K. Poulsen and P.C. Yew. Execution-driven tools for parallel simulation of parallel architectures and applications. In *Proceedings of Supercomputing '93*, pages 860–869, November 1993.

[RBDH97] Mendel Rosenblum, Edouard Bugnion, Scott Devine, and Stephen A. Herrod. Using the SimOS machine simulator to study complex computer systems. *ACM Transactions on Modeling and Computer Simulation*, 7(1):78–103, 1997.

[RBMD03] Mehrdad Reshadi, Nikhil Bansal, Prabhat Mishra, and N. Dutt. An efficient retargetable framework for instruction-set simulation. In *First IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*, pages 13–18, October 2003.

[rei07] Reiser file system home page, 2007. http://www.namesys.com.

[RHWG95] M. Rosenblum, S. A. Herrod, E. Witchel, and A. Gupta. Complete computer system simulation: the SimOS approach. *Parallel & Distributed Technology: Systems & Applications, IEEE*, 3(4):34–43, Winter 1995.

[Rie06] Rolf Riesen. A hybrid MPI simulator. In *2006 IEEE International Conference on Cluster Computing*, pages 1–9, September 2006.

[RKH+08] Michael Resch, Rainer Keller, Valentin Himmler, Bettina Krammer, and Alexander Schulz. The Vampir performance analysis tool-set. In *Proceedings of the 2nd International Workshop on Parallel Tools for High Performance Computing*, pages 139–155. Springer Berlin Heidelberg, 2008.

[RM06] Philip C. Roth and Barton P. Miller. On-line automated performance diagnosis on thousands of processes. In *PPoPP '06: Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 69–80, New York, NY, USA, 2006. ACM.

[RPP01] Miguel Ángel Vega Rodríguez, Juan Manuel Sánchez Pérez, and Juan Antonio Gómez Pulido. An educational tool for testing caches on symmetric multiprocessors. *Microprocessors and Microsystems*, 25(4):187–194, 2001.

[RSJC94] H.A. Rizvi, J.B. Sinclair, J.R. Jump, and J. Carson. Execution-driven simulation of a superscalar processor. In *International Conference on System Sciences. Vol. I: Architecture*, pages 185–194, January 1994.

[RW94] C. Ruemmler and J. Wilkes. An introduction to disk drive modeling. *Computer*, 27(3):17–28, March 1994.

[S.79] Schlesinger S. Terminology for model credibility. *Simulation*, 32(3):103–104, 1979.

[Sar05] Robert G. Sargent. Verification and validation of simulation models. In *WSC '05: Proceedings of the 37th conference on Winter simulation*, pages 130–143. Winter Simulation Conference, 2005.

[SBR05] J. Schnerr, O. Bringmann, and W. Rosenstiel. Cycle accurate binary translation for simulation acceleration in rapid prototyping of SoCs. In *Proceedings of Design, Automation and Test in Europe*, volume 2, pages 792–797, March 2005.

[SCK+93] Richard L. Sites, Anton Chernoff, Matthew B. Kirk, Maurice P. Marks, and Scott G. Robinson. Binary translation. *Communications of ACM*, 36(2):69–81, 1993.

[SCW+02] A. Snavely, L. Carrington, N. Wolter, J. Labarta, R. Badia, and A. Purkayastha. A framework for performance modeling and prediction. In *ACM/IEEE Conference on Supercomputing*, pages 21–21, November 2002.

[SG99] Jiri Schindler and Gregory R. Ganger. Automated disk drive characterization. Technical Report CMU-CS-99-176, School of Computer Science. Carneige Mellon University, Pittsburgh, PA 15213, December 1999.

[Sha98] Robert E. Shannon. Introduction to the art and science of simulation. In *WSC '98: Proceedings of the 30th conference on Winter simulation*, pages 7–14, Los Alamitos, CA, USA, 1998. IEEE Computer Society Press.

[She06] Joel Sherrill. File system simulator. Release 2006-03-16. http://www.omnetpp.org/doc/FSS-doc/neddoc/index.html, 2006.

[SL98] Eric Schnarr and James R. Larus. Fast out-of-order processor simulation using memoization. *SIGPLAN Notices*, 33(11):283–294, 1998.

[SM06] P. Stanley-Marbell. Implementation of a distributed full-system simulation framework as a file system server. In *Proceedings of the 1st International Workshop on Plan 9*, Madrid, Spain, 2006.

[SMW98] Elizabeth Shriver, Arif Merchant, and John Wilkes. An analytic behavior model for disk drives with readahead caches and request reordering. In *SIGMETRICS'98/PERFORMANCE'98, Proceedings of the 1998 ACM SIGMETRICS joint international conference on Measurement and modeling of computer systems*, pages 182–191, New York, NY, USA, 1998. ACM.

[SN05] Jim Smith and Ravi Nair. *Virtual Machines: Versatile Platforms for Systems and Processes*. Morgan Kaufmann, 1st edition, June 2005.

[SPHC02] Timothy Sherwood, Erez Perelman, Greg Hamerly, and Brad Calder. Automatically characterizing large scale program behavior. In *ASPLOS'2002. Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 45–57, 2002.

[SS97] K. A. Smith and M. I. Seltzer. File system aging - increasing the relevance of file system benchmarks. In *Proceedings of the 1997 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, Seattle, Washington, United States, 1997.

[SSV99] David Sundaram-Stukel and Mary K. Vernon. Predictive analysis of a wavefront application using LogGP. *ACM SIGPLAN Notices*, 34(8):141–150, 1999.

[Ste01] Thomas Sterling. An introduction to PC clusters for high performance computing. *International Journal of High Performance Computing Applications*, 15(2):92–101, 2001.

[sys03] SystemC, OSC initiative. Technical report, OSCI, 2003.

[Tan01] Andrew S. Tanenbaum. *MOSS (Modern Operating System Simulators)*. Prencice-Hall, Amsterdam, The Netherlands, second edition, 2001.

[TSG08] S.R. Tarapore, C.W. Smullen, and S. Gurumurthi. MIDAS: An execution-driven simulator for active storage architectures. In *Workshop on Modeling, Benchmarking, and Simulation (Held in Conjunction with ISCA 2008)*, pages 1–10, Beijing, China, June 2008.

[TWL94] C. A. Thekkath, J. Wilkes, and E. D. Lazowska. Techniques for file system simulation. *Software-Practice & Experience*, 24(11):981–999, November 1994.

[UC00] David Ung and Cristina Cifuentes. Machine-adaptable dynamic binary translation. *ACM SIGPLAN Notices*, 35(7):41–51, 2000.

[Ueb97] Christopher W. Ueberhuber. *Numerical Computation 2: Methods, Software, and Analysis*, volume 2. Springer, 1st edition, April 1997.

[Var01] András Varga. The OMNeT++ discrete event simulation system. In *ESM'01: Proceedings of the European Simulation Multiconference*, Prague, Czech Republic, June 2001.

[Var07] András Varga. The INET framework, 2007. http://ctieware.eng.monash.edu.au/twiki/bin/view/Simulation/INETFramework.

[Vog99] Werner Vogels. File system usage in Windows NT 4.0. In *Symposium on Operating Systems Principles*, 1999.

[WAA+04] M. Wang, K. Au, A. Ailamaki, A. Brockwell, C. Faloutsos, and G. R. Ganger. Storage device performance prediction with cart models. In *MASCOTS'04: Proceedings of the 12th Annual Meeting of the IEEE/ACM International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*, Volendam. The Netherlands, October 2004.

[WC06] John Walters and Vipin Chaudhary. Application-level checkpointing techniques for parallel programs. In *Distributed Computing and Internet Technology*, volume 4317 of *Lecture Notes in Computer Science*, pages 221–234. Springer Berlin / Heidelberg, 2006.

[WD05] Norcott W.D. and Capps D. Iozone, August 2005. http://www.iozone.org/.

[WGP09] Xuan Wang and K. Goseva-Popstojanova. Modeling Web request and session level arrivals. In *AINA '09: International Conference on Advanced Information Networking and Applications*, pages 24–32, May 2009.

[WGSS99] Bettina Weiss, Günther Gridling, Ulrich Schmid, and Klaus Schossmaier. The SimUTC fault-tolerant distributed systems simulation toolkit. In *MASCOTS '99: Proceedings of the 7th International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*, pages 68–75, Washington, DC, USA, 1999. IEEE Computer Society.

[WGT+05] David Wang, Brinda Ganesh, Nuengwong Tuaycharoen, Kathleen Baynes, Aamer Jaleel, and Bruce Jacob. DRAMsim: a memory system simulator. *ACM SIGARCH Computer Architecture News*, 33(4):100–107, 2005.

[WHH+00] Harvey Wasserman, Adolfy Hoisie, Adolfy Hoisie, Olaf Lubeck, and Olaf Lubeck. Performance and scalability analysis of teraflop-scale parallel architectures using multidimensional wavefront applications. *The International Journal of High Performance Computing Applications*, 14:330–346, 2000.

[WWFH03a] T.F. Wenisch, R.E. Wunderlich, B. Falsafi, and J.C. Hoe. Applying SMARTS to SPEC CPU2000. Technical Report 2003-1, Computer Architecture Lab at Carnegie Mellon, April 2003.

[WWFH03b] R.E. Wunderlich, T.F. Wenisch, B. Falsafi, and J.C. Hoe. SMARTS: accelerating microarchitecture simulation via rigorous statistical sampling. In *30th Annual International Symposium on Computer Architecture*, pages 84–95, June 2003.

[WWFH05] Thomas F. Wenisch, Roland E. Wunderlich, Babak Falsafi, and James C. Hoe. TurboSMARTS: accurate microarchitecture simulation sampling in minutes. In *SIGMETRICS'05: Proceedings of the International Conference on Measurements and Modeling of Computer Systems*, pages 408–409, June 2005.

[XWHC05] Jiang Xu, Wayne Wolf, Joerg Henkel, and Srimat Chakradhar. A methodology for design, modeling, and analysis of networks-on-chip. In *ISCAS'05: IEEE International Symposium on Circuits and Systems.*, volume 2, pages 1778–1781, Princeton University, May 2005.

[Zei84] Bernard Phillip Zeigler. *Theory of Modelling and Simulation.* Krieger Publishing Company, 1984.

[Zhu09] Colin (Lin) Zhu. Data center storage networking simulation, 2009. SimSANs version 3 is available at http://www.simsans.org.

[ZKK04] G. Zheng, Gunavardhan Kakulapati, and L.V. Kale. BigSim: a parallel simulator for performance prediction of extremely large parallel machines. In *IPDPS'04: Parallel and Distributed Processing Symposium*, pages 78–87, April 2004.